tigerjython

Jarka Arnold, Tobias Kohn, Aegidius Plüss

CONCEPTS DE PROGRAMMATION

en Python avec la plateforme TigerJython

Le manuel interactif en ligne offre de nombreuses idées de cours d'informatique avec de nombreux exemples de programmes complets sur des sujets d'actualité. L'objectif n'est pas d'introduire le langage de programmation Python, mais plutôt de connaître les concepts de base de la programmation qui sont de plus en plus importants dans la vie quotidienne et dans les écoles secondaires.

www.programmierkonzepte.ch/franz

Version 2.7, Dernière mise à jour: nov 24. 2021

Contenu

1. INTRODUCTION 7
1.1 Installation 8
1.2 Premiers pas 13
1.3 Instructions à l'attention de l'enseignant 17
1.4 Raspberry PI 17

2. TORTUE GRAPHIQUE 22
2.1 Déplacer une tortue 23
2.2 Utiliser des couleurs 26
2.3 Répétition 29
2.4 Fonctions 33
2.5 Paramètres 36
2.6 Variables 39
2.7 Branchements conditionnels (Selection) 42
2.8 Boucles while 47
2.9 Récursions 52
2.10 Contrôle par les événements 57
2.11 Objets tortues 63
2.12 Impression 69
2.13 Documentation des graphiques tortue 72

3. GRAPHISME & IMAGES 79
3.1 Coordonnées 80
3.2 Boucles for 84
3.3 Programmation structurée 88
3.4 Fonctions avec valeur de retour 91
3.5 Variables globales et animations 94
3.6 Contrôle du programme au clavier 98
3.7 Événements de la souris 96
3.8 Art filaire 102
3.9 Listes 110
3.10 Hasard et nombres aléatoires 120
3.11 Traitement d'images 127
3.12 Impression d'images 137
3.13 Widgets 140
3.14 Documentation GPanel 145

4. SON 154
4.1 Restituer un son 155
4.2 Édition de son 159
4.3 Enregistrer des sons 162
4.4 Synthèse vocale 165
4.5 Expériences acoustiques 170
4.6. Documentation du système sonore 173

5. ROBOTIQUE 175
5.1 Mode réel et mode simulation 176
5.2 Robots intelligents 184
5.3 Contrôle et régulation 194
5.4 Technologie des capteurs 198
5.5 Documentation des modules de robotique 206

6. INTERNET 211
6.1 Html, chaînes de caractères 212
6.2 Modèle client-serveur, protocole HTTP 217
6.3 Recherche Bing, dictionnaires 223

7. JEUX & POO 228
7. 1. Des objets, encore des objets 229
7.2 Classes et objets 235
7.3 Jeux d'arcade, Frogger 244
7.4 Grille de jeu, jeu de plateau, Solitaire 252
7.5 Sprites animés 259
7.6 Méthodes principales de la classe JGameGrid 272

8. EXPÉRIENCE INFORMATIQUES 277
8.1 Simulations 278
8.2 Populations 282
8.3 Hypothèses et tests statistiques 296
8.4 Temps moyen d'attente 303
8.5 Suites et convergence 316
8.6 Corrélation, régression 323
8.7 Nombres complexes & fractales 340
8.8 Analyse spectrale 352
8.9 Dynamique de groupe 358
8.10 Marche aléatoire 366

9. BASES DE DONNÉES & SQL 373
9.1 Persistance, fichiers 374
9.2 Bases de données en ligne 379
9.3 Système de réservation en ligne 388
9.4 Requêtes SQL 394

10. EFFICACITÉ & LIMITATIONS 395
10.1 Complexité de tri 396
10.2 Problèmes insolubles 404
10.3 Retour sur trace (backtracking) 411
10.4 Plus court chemin, problème des 3 récipients 422
10.5 Cryptosystèmes 431
10.6 Automates finis 439
10.7 Information et ordre 448

11. ANNEXES 456
11.1 Jeux logiques amusants 457
11.2 Trappes, règles et astuces 471
11.3 Bogues et déboguage 480
11.4 Traitement parallèle 489
11.5 Interface série 503
11.6 Sockets TCP 506

12. CONTACT 521

Version 1.0, Juillet 2016

Auteurs: Jarka Arnold, Tobias Kohn, Aegidius Plüss
Traduction française: Cédric Donner
Contact: help@tigerjython.com

AVANT-PROPOS

Au début des années 1950, j'ai eu le privilège d'utiliser le premier ordinateur programmable disponible en Suisse, le Zuse 4, pour rédiger ma thèse de doctorat à l'École Polytechnique Fédérale de Zürich (ETHZ). Les premiers pas de notre pays dans le domaine des sciences computationnelles, combinées par la suite sous le vocable « informatique », n'ont été faits dans les universités cantonales que très progressivement et il a fallu attendre longtemps pour que l'informatique soit reconnue comme une discipline scientifique à part entière. À l'ETHZ, ce n'est qu'en 1974 que les professeurs des sciences computationnelles ont pu obtenir la création de l'institut d'informatique et il a fallu attendre 1981 pour qu'un département d'informatique à proprement parler voie le jour.

Le développement rapide de la performance et de la miniaturisation des ordinateurs a contribué de manière significative à une augmentation fulgurante de la production de données. Cela a engendré une expansion massive des moyens d'échanges de données qui n'a pu elle-même être rendue possible que par un usage encore plus répandu des ordinateurs. En conséquence, il a fallu améliorer les techniques de communication et démocratiser largement leur usage.

Au sein de notre planète menacée par une croissance constante des populations et leur aspiration à de meilleures conditions de vie, la Suisse ne peut maintenir son développement exceptionnels, son niveau de vie très élevé et son système de démocratie directe que si elle peut compter sur un système éducatif moderne et efficace doté de recherche de très haute qualité. Il faut donc, pour garantir la progression de nos universités, non seulement tenir compte des derniers développements dans le domaine des TIC, mais également repenser les bases de l'éducation dispensée dans les écoles primaires et secondaires ainsi que l'enseignement de l'informatique dans les écoles de maturité (gymnases). Les trois compétences fondamentales de lecture, d'écriture et d'arithmétique ne sont plus suffisantes pour garantir une existence satisfaisante dans notre monde actuel, tant les ordinateurs y jouent un rôle important dans la vie personnelle ainsi que professionnelle. En tant qu'ancien directeur de l'Office fédéral de l'éducation et de la science, je me suis battu pour établir l'informatique comme une branche à part entière dans les écoles de maturité.

La grande question était de savoir si l'intégration des TIC dans les programmes d'enseignement était suffisante ou s'il était nécessaire de dispenser en informatique un enseignement de connaissances plus approfondies qui permettraient une utilisation plus rationnelle des technologies de l'information modernes. Les auteurs de la plateforme éducative *TigerJython* donnent une réponse très concrète à cette question. Cette plateforme montre en effet comment les concepts les plus importants de l'informatique peuvent être enseignés de manière accessible en utilisant le langage de programmation Python et un environnement de programmation conçu dans un souci pédagogique. Cette contribution fournit ainsi une justification crédible à leur recommandation d'introduire le sujet de l'informatique dès la sixième année d'école primaire. Le récent article de presse intitulé "Die Schweizer EGovernment-Angebote sind im internationalen Vergleich nur Mittelmass... Die Schweiz ist unter den europäischen Staaten gar auf den vorletzten Platz zurückgefallen" montre, à mon avis, le besoin urgent d'une réponse appropriée à cette proposition puisque cette régression alarmante est avant tout due à une connaissance inadéquate de l'informatique dans les institutions éducatives. De plus, il n'y a dans notre pays riche, pourvu de l'une des plus hautes densités d'ordinateurs au monde, aucun manque matériel qui empêcherait de redresser la barre.

Prof. Dr. sc. math., Dr. h.c. Urs Hochstrasser, ancien directeur de l'Office fédéral de l'éducation et de la science (**http://hochstrasserurs.blogspot.ch**)

PRÉFACE

TigerJython est constitué de moyens d'éducation en ligne ainsi que d'un environnement de développement spécialement conçu pour l'éducation. Ce moyen d'enseignement débute avec la tortue graphique et poursuit ensuite avec des sujets aussi divers que la programmation de robots Lego, la manipulation de données multimédia, le développement de jeux vidéo, les bases de données et les simulations stochastiques. En raison de sa structure modulaire et de ses nombreux exemples et exercices, TigerJython se prête aussi bien à l'utilisation en classe qu'à l'étude en autodidacte. Les premiers chapitres peuvent déjà être utilisés dans un cours d'introduction à l'informatique à l'école obligatoire (niveau S1). L'ouvrage, pris dans son ensemble, couvre, par le choix des sujets et la profondeur de la matière, le contenu que devrait aborder un cours d'informatique fondamentale au gymnase.

Nous sommes convaincus que l'enseignement des concepts fondamentaux d'informatique contribue de manière essentielle au développement intellectuel des adolescents. Selon nous, l'informatique devrait être déjà enseignée à l'école primaire dès l'âge de 12 ou 13 ans dans le but d'éveiller très tôt, chez les élèves, un intérêt et une passion pour la pensée logique et technique.

La première édition du présent moyen d'enseignement fut développée en 2013. La seconde édition actuelle intègre de nombreuses révisions et corrections. Nous avons incorporé dans ce matériel nos années d'expérience dans l'enseignement de l'informatique à nos étudiants et aux enseignants d'informatique. Notre intention a toujours été, en plus de développer auprès des jeunes gens et jeunes filles l'intérêt et le plaisir pour la pensée algorithmique, de soutenir les enseignants dans cette mission.

Ce matériel d'enseignement cherche à réduire au maximum, par l'utilisation de l'environnement de programmation *TigerJython* et le langage de programmation *Python*, tout obstacle susceptible d'empêcher les premiers pas en programmation. Tout le contenu de ce matériel d'enseignement est pour ainsi dire issu d'un même moule. La plus grande partie du contenu prend racine dans des situations quotidiennes concrètes ainsi que dans les situations problèmes surgissant dans d'autres branches scolaires. De cette manière, les connaissances d'informatique peuvent également être appliquées à d'autres disciplines.

Bien que le langage Python ait été développé à l'origine par le hollandais Guido van Rossum il y a déjà 20 ans en arrière, ce n'est que récemment qu'il a fait son entrée dans les écoles jusqu'à constituer actuellement une véritable tendance dans de nombreuses institutions de formation renommées. Cela provient probablement du fait que Python, en tant que langage interprété disposant d'un espace de nom global, est d'un apprentissage très aisé mais également du fait qu'il ne requière que très peu de ressources machine à tel point qu'il tourne même sur des micro-systèmes. De plus, avec *TigerJython*, nous offrons un environnement de développement particulièrement bien adapté aux étudiants vu le bon équilibre entre la simplicité et le professionnalisme qui le caractérise. De notre avis, il est particulièrement bien adapté pour les cours d'informatique pour les raisons suivantes :

- L'installation sur un poste Windows/Mac/Linux ne nécessite que la copie d'un seul et unique fichier dans n'importe quel dossier de la machine accessible en écriture. Les enseignants peuvent ainsi immédiatement commencer à enseigner, même sur des postes pour lesquels ils n'ont aucun privilège administrateur.
- L'EDI (IDE) de *TigerJython* est si intuitif que son utilisation ne nécessite absolument aucune instruction préalable. En particulier, il n'y a aucune nécessité de créer un projet.
- *TigerJython* effectue une analyse très précise des erreurs d'exécution du programme et affiche des messages d'erreurs compréhensibles même par des programmeurs complètement débutants.
- *TigerJython* contient de nombreux modules additionnels spécialement adaptés pour l'enseignement tels que la tortue graphique, les graphiques en coordonnées, la robotique ou la programmation de jeux vidéo.

Nous espérons parvenir à transmettre avec TigerJython et ce matériel d'enseignement une part de notre enthousiasme pour l'enseignement de la science informatique.

Remerciements:
Nous tenons à remercier spécialement toutes les personnes qui ont contribué au succès de TigerJython par leurs suggestionset commentaires, notamment Walter Gander (ETH Zürich), Juraj Hromkovic (ETH Zürich), Theo Heußer (Gymnase de Hemsbach), Urs Hochstrasser (ancien président de l'Office fédéral pour l'enseignement et la science, Berne).

Nous remercions également Kristin et Florian Thalmann pour la traduction anglaise et Cédric Donner pour la traduction française de l'ouvrage originale en langue allemande.

Juillet 2016. Jarka Arnold, Tobias Kohn, Aegidius Plüss

Förderungsprojekt Nr. 2015-025 der Schweizerischen Akademie der Technischen Wissenschaften an die Schweizer Informatikgesellschaft

chapter one

LEARNING ENVIRONMENT

Objectifs d'apprentissage

- Installer l'environnement de développement TigerJython sur un ordinateur.
- Être capable d'éditer et exécuter un programme.
- Être capable de modifier les réglages de l'environnement.
- Être capable d'utiliser la console pour effectuer des calculs simples.
- Être conscient qu'il est possible d'utiliser TigerJython sur un Raspberry Pi .

"I think everybody in this country should learn how to program a computer because it teaches you how to think."

Steve Jobs, The lost interview

1.1 INSTALLATION

INTRODUCTION

L'environnement de développement TigerJython est bien adapté pour les programmeurs novices et pour les utilisateurs travaillant dans un environnement protégé comme des postes sur lesquels les droits sont limités. La distribution de TigerJython tient entièrement dans une archive JAR téléchargeable gratuitement.

Télécharger TigerJython

La distribution contient tous les composants nécessaires pour programmer, hormis l'environnement d'exécution Java (**JRE** = Java Runtime Environment). TigerJython est même capable de s'exécuter depuis un support de stockage externe tel qu'une clé USB.

TigerJython est indépendant de la plateforme et tourne sans problème sous Windows, Mac OS, Linux, et même sur un Raspberry Pi.

INSTALLATION

Pour installer TigerJython, télécharger le fichier *tigerjython2.jar* et l'enregistrer en local sur le disque dur. Il est recommandé de créer un raccourci sur le bureau pour pouvoir démarrer l'environnement facilement. Sur Linux, il est nécessaire de donner les droits d'exécution sur le fichier JAR avec la commande *chmod*. Ensuite, on peut sauver tous les programmes Python dans le même dossier que l'archive JAR.

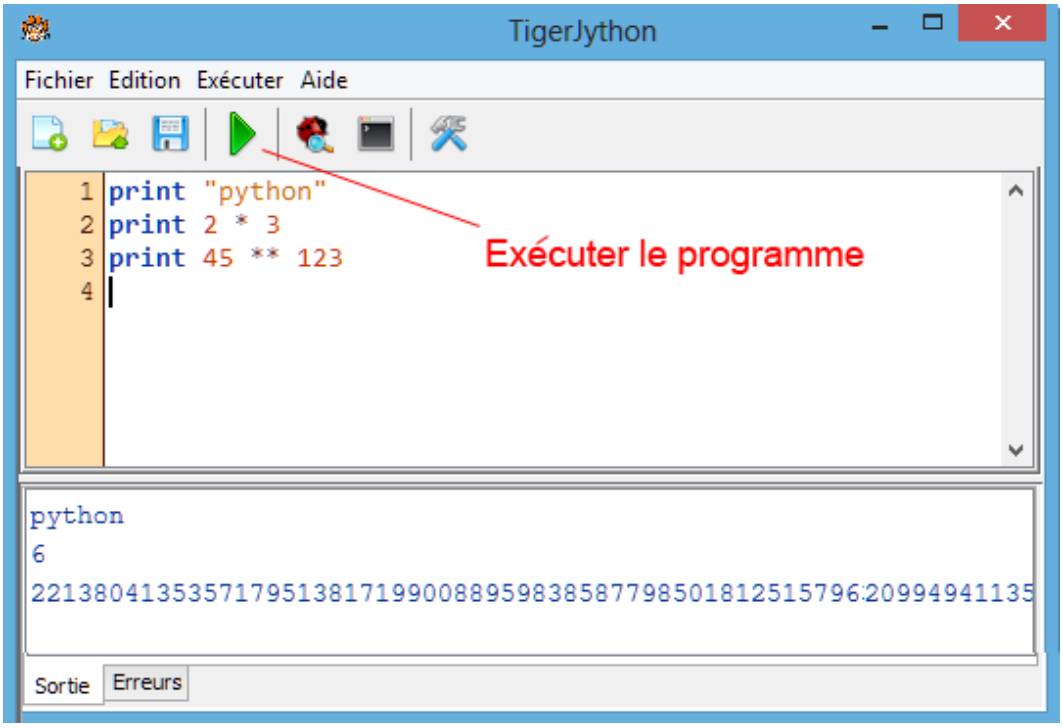
Pour assigner un joli icône au raccourci, il est possible de le télécharger **ici** pour Windows, **ici** pour Mac and **ici** pour Mac/Linux.

Remarque : il est possible que le démarrage par double clic sur l'archive *tigerjython2.jar* ne fonctionne pas correctement si l'application par défaut pour ouvrir les fichiers portant l'extension JAR n'est pas paramétrée sur Java. Cela peut arriver parfois si un programme de compression (par exemple WinRAR) a modifié les paramètres d'application par défaut pour ouvrir les fichiers JAR. Dans ce cas, il faut associer l'environnement d'exécution Java avec les fichiers .JAR de la manière suivante :

- Faire un clic droit sur le fichier *tigerjython2.jar* et cliquer sur l'option *ouvrir avec …*
- Dans le menu contextuel, choisir *ouvrir avec une autre application*
- Dans la boîte de dialogue, sélectionner l'application *Java(TM) Plateform SE Binary* ou similaire tout en cochant l'option *toujours utiliser cette application pour ouvrir les fichiers .jar*. Si l'application *Java* n'apparaît pas dans la liste des applications disponibles, il faut réinstaller l'environnement d'exécution Java (JRE).

PREMIERS PAS

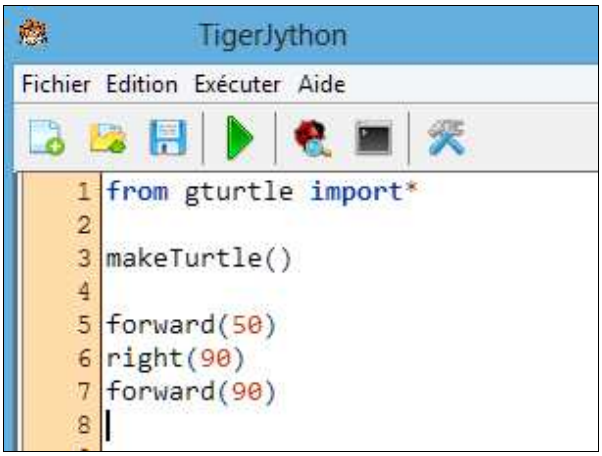
Démarrer l'éditeur TigerJython en cliquant sur l'archive ***tigerjython2.jar*** ou sur le lien pointant vers l'archive. L'éditeur est très intuitif à prendre en mains. La barre d'outils présente des boutons permettant de *Créer un nouveau document (programme), Ouvrir, Enregistrer, Exécuter, Régler le débogage sur on/off, Afficher la console* et *Réglages.*



Il ne faut pas hésiter à tester ces fonctionnalités en saisissant quelques instructions *print* et cliquer ensuite sur le bouton vert *Exécuter le programme*. Au contraire de la plupart des langages de programmation, la taille des nombres que Python est capable de traiter n'est limité que par les capacités de la machine, ce qui lui permet de bien traiter le nombre gigantesque « 45 exposant 123 » qui n'est affiché que partiellement dans la capture d'écran ci-dessus.

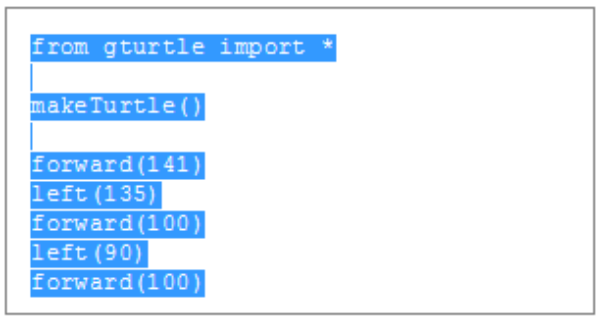
ÉDITER UN PROGRAMME

Pour commencer, écrire un programme tout simple qui va créer un graphique à l'aide de la tortue.



Lors de l'édition, il ne faut pas hésiter à faire un usage abondant des raccourcis standards:

Ctrl+C	Copier
Ctrl+V	Coller
Ctrl+X	Couper
Ctrl+A	Tout sélectionner
Ctrl+Z	Annuler
Ctrl+S	Enregistrer
Ctrl+N	Nouveau document
Ctrl+O	Ouvrir
Ctrl+Y	Refaire (contraire de l'annulation)
Ctrl+F	Chercher
Ctrl+H	Chercher et remplacer
Ctrl+Q	Commenter/décommenter les lignes
Ctrl+D	Supprimer la ligne



Sélectionner le code source (Ctrl+C, Ctrl+V

Les programmes exemples utilisés dans le tutorial sont choisis pour être facilement utilisés comme modèles de base à partir desquels construire tes propres programmes. Sur le site, il est possible de sélectionner tout le code d'un programme en cliquant sur *Sélectionner tout le code*. Il est également possible de sélectionner une partie du code à la souris et de le copier avec Ctrl+C pour le coller ensuite dans l'éditeur avec le raccourci Ctrl+V.

L'instruction **import** précise que l'on doit charger ces instructions avant de pouvoir les utiliser. L'instruction **makeTurtle()** créer une fenêtre contenant une tortue que ton programme va contrôler. Les lignes **lignes suivantes** présentent les instructions exécutées par la tortue à proprement parler.

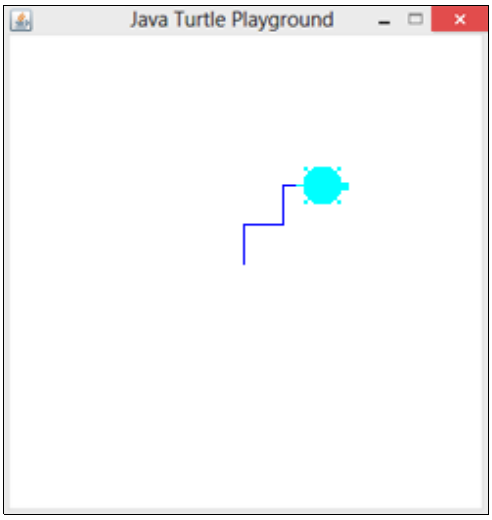
```
from gturtle import *

makeTurtle()

forward(141)
left(135)
forward(100)
left(90)
forward(100)
```

Sélectionner le code source (Ctrl+C pour copier, Ctrl+V

La « sélection magique » facilite le repérage dans le code source des programmes des instructions mentionnées dans le texte explicatif.

En cliquant sur les mots en vert, les instructions correspondantes sont mises en évidence dans le programme.

EXÉCUTER LE PROGRAMME



Pour exécuter le programme, cliquer sur le triangle vert.

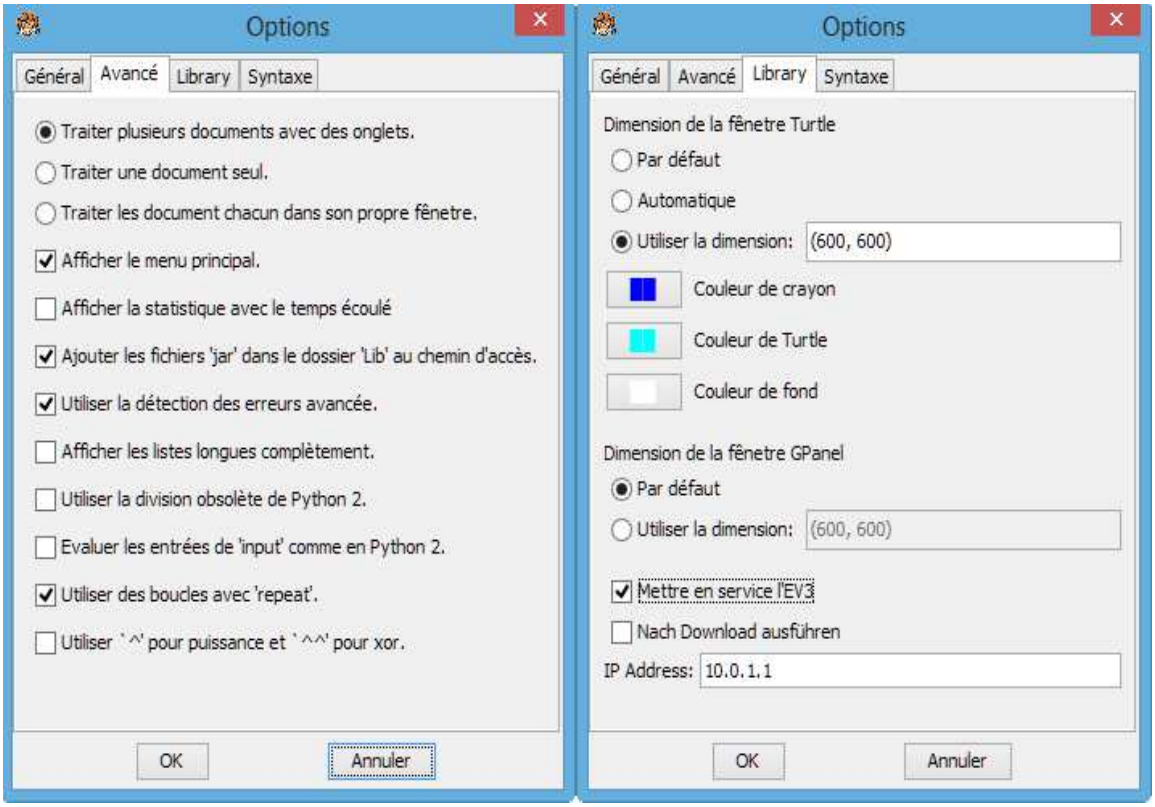
Le graphique engendré par le programme apparaît dans une nouvelle fenêtre.

S'il y a un problème avec le programme, des messages d'erreur vont apparaître dans la fenêtre *Problems*.

PARAMÈTRES DE CONFIGURATION

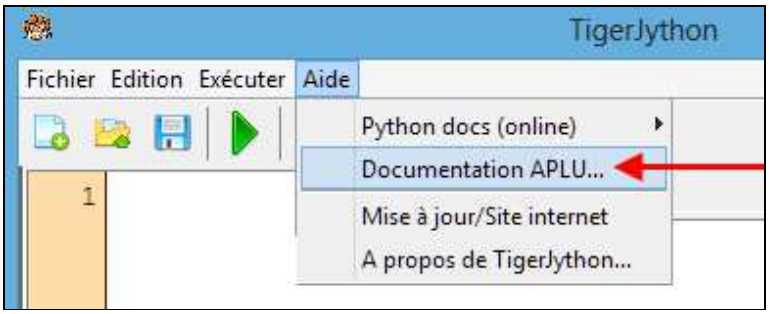
Voici les réglages qu'il est possible d'effectuer en cliquant sur le bouton ci-contre.

- Taille et couleur de la police ainsi que la taille des indentations
- Langue de l'interface (Allemand, Anglais, Français, Italien)
- Couleur de fond et taille par défaut de la fenêtre tortue, du crayon et de la tortue
- Outils supplémentaires pour permettre la programmation d'un robot EV3



DOCUMENTATION

TigerJython regorge de modules supplémentaires intégrés, par exemple pour les graphiques de tortues. Il est possible d'en visualiser la documentation en cliquant sur *APLU Documentation* dans le menu *Aide*.



EXEMPLES

Il est recommandé de parcourir le matériel d'enseignement chapitre par chapitre et de tester chaque programme proposé de manière individuelle dans TigerJython. Pour cela, il suffit de les sélectionner sur le site, de les copier avec *Ctrl+C*, les coller dans TigerJython avec *Ctrl+V* et, finalement, de les enregistrer avec un nom approprié portant l'extension *.py. Il est alors possible de les exécuter.

Tous les programmes sont téléchargeables ici.

INSTALLATION MULTI-UTILISATEURS EN LABORATOIRE

TigerJython étant limité à une seule archive JAR *tigerjython2.jar*, il est très facile de le supprimer d'un poste. Aucun processus d'installation particulier n'est nécessaire et le registre Windows n'est pas affecté. Pour stocker les réglages utilisateurs, un fichier de configuration nommé *tigerjython2.cfg* est créé dans le dossier contenant l'archive JAR. Dans un

environnement multi-utilisateurs, ce fichier peut être géré par un administrateur système. Pour plus d'informations, se reporter à **cett page**.

Remarque: Dans de très rares cas, les archives JAR des bibliothèques APLU (par exemple aplu5.jar) sont copiées dans le dossier *<jrehome>/lib/ext*, ce qui peut occasionner des conflits avec TigerJython qui utilise des versions de ces bibliothèques configurées de manière spéciale.

LANCEUR POUR LE BUREAU UBUNTU

Télécharger l'image tjlogo64.png depuis **cet emplacement**et copier le fichier dans le dossier contenant *tigerjython2.jar*.

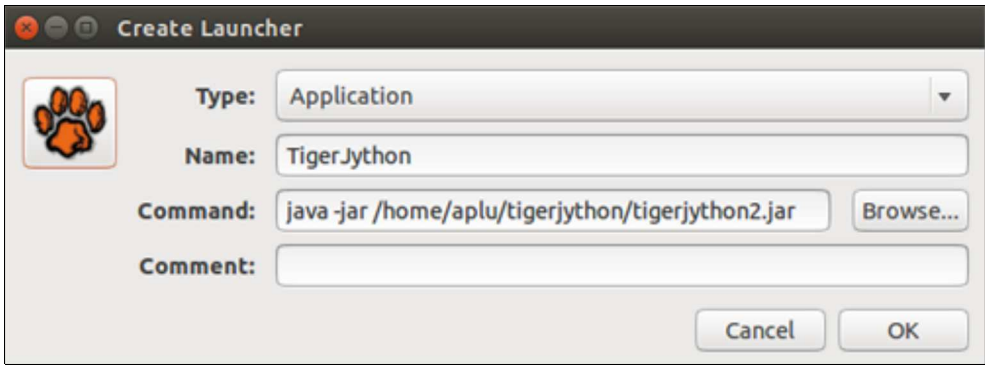
Pour les nouvelles versions de Ubuntu, il faut installer le paquet gnome-panel avec la commande:

```
sudo apt-get install gnome-panel
```

Pour générer le lanceur, faire Alt-F2 et saisir la commande

```
gnome-desktop-item-edit --create-new ~/Desktop
```

et compléter boîte de dialogue en ajustant le chemin vers *tigerjython2.jar* :



Cliquer sur l'icône et spécifier l'image *tjlogo64.png* téléchargée précédemment puis confirmer avec OK.

DÉMARRER UN PROGRAMME SANS L'IDE TYGERJYTHON

Du fait que Python soit un langage interprété, il est nécessaire que l'interpréteur Python soit démarré pour exécuter un programme. Sous Windows, il est possible de lancer l'exécution d'un script Python à l'aide de la ligne de commande

java -jar jython.jar <prog.py>

qui fonctionnera pour autant que le dossier courant contienne l'archive de l'interpréteur jython.jar ainsi que le script *prog.*py.

Pour permettre aux modules additionnels de la bibliothèque APLU de fonctionner sans encombres et d'être préchargés automatiquement par l'interpréteur, ceux-ci doivent être inclus dans l'archive JAR de l'interpréteur. Une archive JAR modifiée nommée ajython.jar contenant ces modules est disponible en téléchargement **ici**. Le fichier *readme.txt* présent dans cette archive donne plus d'indication sur la procédure à suivre. Il faut cependant être conscient que certaines spécificités du langage TigerJython, telles que la structure *repeat* et certaines boîtes de dialogue, ne sont pas disponibles dans l'interpréteur Jython standard. Cependant, il n'est pas nécessaire que Python ou Jython soit installé.

1.2 PREMIERS PAS

INTRODUCTION

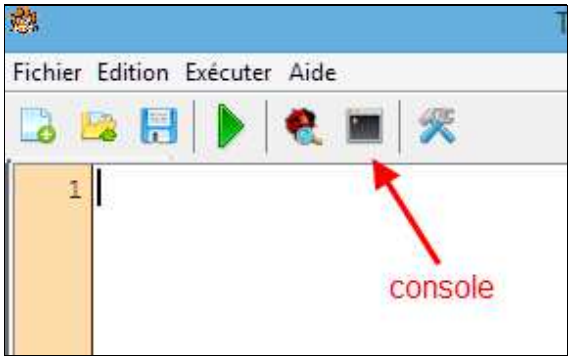
Un programme informatique généralement en une suite d'instructions. Python permet d'exécuter de manière instantanée et interactive des instructions individuelles. Cela convient particulièrement bien pour une première utilisation de Python ou tester rapidement une idée. Pour cela, il faut cliquer sur l'icône de la console, ce qui va ouvrir la fenêtre de la console. Celle-ci permet de saisir une instruction sur la ligne débutant par « >>> » qui sera exécutée avec la touche ENTER (Retour chariot). Comme dans tout éditeur, il est possible de naviguer à l'intérieur de la ligne à la souris ou au clavier à l'aide des touches directionnelles et d'insérer ou supprimer des caractères individuels. Aussitôt que la touche ENTER est enfoncée, la ligne d'instruction est exécutée, à moins qu'il s'agisse d'une instruction s'étendant sur plusieurs lignes. Dans ce dernier cas, lorsque toutes les lignes ont été saisies, il faut appuyer deux fois sur ENTER pour exécuter l'instruction saisie.

Il est possible de sélectionner les instructions déjà exécutées avec la souris et les copier avec *Ctrl+C*. On peut ensuite les coller le contenu du presse-papier sur la ligne de commande à exécuter avec *Ctrl+V*.

Il est possible également de naviguer dans l'historique des instructions exécutées avec les touches directionnelles haut et bas du clavier. Le caractère souligné représente le résultat de l'évaluation de la dernière instruction.

FAIRE CONNAISSANCE DE PYTHON

Voici quelques propositions d'instructions à exécuter. Laissez libre cours à votre imagination pour en tester d'autres. Démarrer TigerJython et cliquer le bouton *Console*.



Saisir les exemples ci-dessous dans la console pour observer comment Python effectue les quatre opérations arithmétiques de base:

```
>>> 4 + 13
    17

>>> 2.5 - 5.7
    -3.2

>>> 1356 * 22345
    30299820

>>> 1 / 7
    0.14285714285714285
```

Comme le montrent les exemples précédents, on peut utiliser des nombres entiers au décimaux. Les nombres entiers sont appelés *integer (int)* et les décimaux sont appelés *float*.

On peut exécuter plusieurs opérations dans une même instruction pour former des expressions plus complexes, comme en mathématiques. Il faut bien faire attention à la précédence des

opérations qui s'applique comme en mathématiques, où * et / ont une priorité supérieure (on dit « précédence » en termes techniques) aux opérateurs + et -. Pour des opérations de même priorité, l'expression est évaluée de gauche à droite. Il est possible de rajouter des parenthèses dans l'expression comme en mathématiques pour regrouper les opérations qui sont à évaluer ensemble. Attention cependant à une petite différence d'avec la notation algébrique standard : il n'est possible d'utiliser que les parenthèses « rondes » () et non les parenthèses « carrées » [] qui ont une signification différente pour Python.

```
>>> (66 - 12) * 5 / 6
    45.0

>>> 66 - 12 * 5 / 6
    56.0
```

La **division entière** et le **reste**(opérateur **modulo**) sont également importants :

```
>>> 5 // 3
    1

>>> 5 % 3
    2
```

Python peut gérer sans problème des nombres très grands survenant par exemple avec l'opérateur d'exponentiation **

```
>>> 45**123
    221380413535717951381719900889598385877985018125157960
    354952620994941135358805405606080888944357204960582620
    340773786668272890150812708415152294926874897612813 7
    612813664505432287299413474102038890123367309570312 5L
```

Es gibt eine Reihe von eingebauten Funktionen, beispielsweise:

```
>>> abs(-9)
    9

>>> max(1, 5, 2, 4)
    5
```

D'autres fonctions sont disponibles après l'importation des modules externes respectifs. Il est possible d'importer de deux manières différentes des modules supplémentaires. Avec la première variante (à gauche), l'appel de la fonction se fera en faisant précéder le nom de la fonction par le nom du module qui la définit. Dans la deuxième variante (à droite), les fonctions sont importées dans l'espace de nom global et peuvent être appelées directement sans les précéder du nom du module. Dans un très court programme, la deuxième version convient très bien mais on préfère généralement la première variante dans les projets plus complexe.

```
>>> import math
>>> math.pi
    3.141592653589793
>>> math.cos(pi)
    -1
```

```
>>> from math import *
>>> pi
    3.141592653589793
>>> sin(pi)
    1.2246467991473533e-16
```

On peut observer au passage que l'ordinateur ne calcule jamais de manière exacte puisque *sin(pi)* vaut théoriquement exactement 0.

Une suite de caractères (lettres, ponctuation, …) est appelée ***chaine de caractères*** (*string* en anglais) et on peut les définir en utilisant des guillemets simples ou doubles. Il est possible d'écrire des chaînes de caractères et d'autres valeurs vers la fenêtre de sortie à l'aide de la commande *print*. La virgule permet de séparer plusieurs valeurs à afficher. L'instruction
`>>> print "Le résultat est", 2 ,` Produit la sortie suivante dans la fenêtre

.

Comme en mathématiques, on peut assigner des valeurs à des variables. Pour ce faire, on utilise un ***identifiant*** (*identifier* en anglais) comportant une ou plusieurs lettres. Certains caractères ne sont pas autorisés pour les identifiants tels que les espaces, les guillemets, les caractères accentués ou tout autre caractère spécial.

Un des avantages des variables est de stocker le résultat d'un calcul précédemment effectué pour éviter de devoir le refaire plus tard. Le panneau droit de la console TigerJython est fort pratique puisqu'il affiche les identifiants connus. Les variables définies dans la ligne de commande sont visibles dans l'espace de nom intitulé *Globals*.

```
>>> a = 2
>>> b = 3
>>> sum = a + b
>>> print "La somme de", a, "et", b, "est", sum
```

Produit la sortie suivante : La somme de sum de 2 et 3 est 5

Une collection ordonnée d'éléments de type quelconque est nommée une ***liste***. Les listes constituent un type de données extrêmement utile et incontournable dans tous les langages de programmation (d'autres langages appellent ce type de données des ***tableaux***). En Python, on définit une liste en énumérant ses différents éléments séparés par une virgule et entre crochets carrés. On peut également afficher les listes dans la sortie du programme avec l'instruction *print*.

```
>>> li = [1, "chicken", 3.14]
>>> print li
```

In the output window: `[1, "chicken", 3.14]`

Les listes et d'autres objets possèdent des fonctions associées appelées méthodes. Celles-ci permettent d'agir sur les objets auxquels elles sont rattachées. Il est par exemple possible d'ajouter de nouveaux éléments à la fin d'une liste à l'aide de la méthode ***append()***.

```
>>> li.append("egg")
>>> print li
```

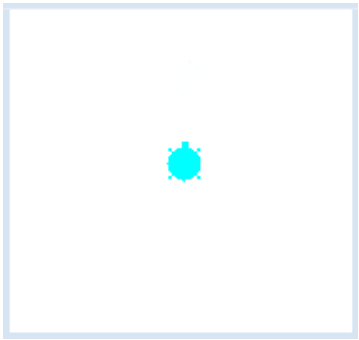
Ce qui produit la sortie: `[1, 'chicken', 3.14, 'egg']`

Il est également possible de définir ses propres fonctions. Pour ce faire, on utilise le mot-clé *def*. On indique la valeur de retour de la fonction à l'aide du mot-clé *return*. Après qu'une fonction a été définie, il est possible de l'appeler de la même manière que les fonctions prédéfinies par Python :

```
>>> def sum(a, b):
>>>     return a + b
>>> sum(2, 3)
    5
```

ENVOYER DES COMMANDES À LA TORTUE

La console est fort utile pour tester rapidement quelques commandes ou fonctions. Par exemple, pour se familiariser avec le module de graphique tortues, il faut d'abord importer le module *gturtle* et créer ensuite une fenêtre comportant une tortue avec la commande *makeTurtle()*.



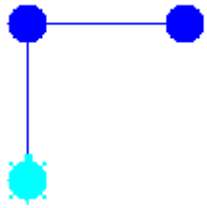
```
>>> from gturtle import *
>>> makeTurtle()
```

Suite à ces instructions, toutes les commandes comprises par la tortue sont à disposition. Par exemple :

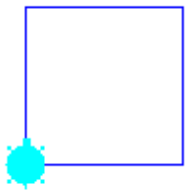
forward(100)	Raccourci: fd(100)	Avancer la tortue de 100 pixels
back(50)	Raccourc: bk(50)	Reculer la tortue de 50 pixels
left(90)	Raccourc: lt(90)	Tourner de 90° vers la gauche
right(90)	Raccourc: rt(90)	Tourner de 90° vers la droite
clearScreen()	Raccourc: cs()	Effacer toutes les traces et replacer la tortue au centre de la fenêtre

Exemple::

```
>>> fd(100)
>>> dot(20)
>>> rt(90)
>>> fd(100)
>>> dot(20)
>>> home()
```

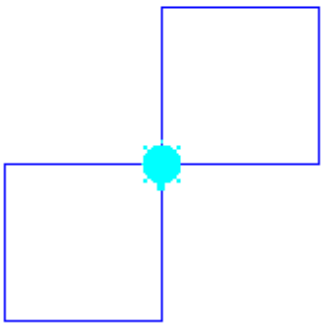


Avec le mot-clé *repeat*, on peut exécuter plusieurs fois de manière répétée une ou plusieurs instructions. Pour exécuter de manière répétée une suite de commandes en tant que bloc d'instructions, il faut qu'elles soient toutes indentées de manière cohérente.



```
>>> repeat 4:
        fd(100)
        rt(90)
```

Comme le montre un exemple précédent, il est possible de combiner plusieurs instructions sous un nom personnalisé en définissant une fonction. L'avantage principal des fonctions est qu'elles permettent d'exécuter aussi souvent que désiré plusieurs instructions à l'aide du nom de cette fonction. Cela permet d'éviter de retaper les mêmes suites d'instructions à plusieurs endroits du programme.



```
>>> def drawSquare():
      repeat 4:
         fd(100)
         rt(90)

>>> drawSquare()
>>> rt(180)
>>> drawSquare()
```

Pour bien apprendre la programmation, il ne faut pas hésiter à essayer par soi-même en essayant pour le plaisir d'autres instructions que celles proposées dans le texte. On peut trouver un aperçu des différentes commandes comprises par la tortue dans la r **documentation du module gTurtle**. Cette documentation fait partie d'un chapitre qui montre de manière systématique comment développer des programmes complets.

1.3 INSTRUCTIONS À L'ATTENTION DE L'ENSEIGNANT

Le présent cours procède d'une méthodologie interne qui passe du simple au complexe. Chaque chapitre est ainsi construit sur les connaissances et compétences développées dans les chapitres précédents. Au total, le matériel du présent cours couvre environ deux à trois ans de cours de base en programmation. Suivant le niveau de la classe et la dotation horaire du cours, il est possible de ne suivre que certains chapitres et d'en laisser tomber d'autres. Les concepts et termes non abordés devront alors être traités de manière indépendante. Puisque les graphiques avec la tortue constituent une manière très intéressante d'introduire les notions fondamentales, le chapitre portant sur ce sujet introduit la plupart des notions fondamentales absolument incontournables. D'après notre expérience d'enseignement, nous suggérons les **variantes minimales** suivantes .

1. Graphiques avec la tortue et projet d'étudiants (en guise d'introduction à la science informatique aux degrés d'enseignement S1 et S2 pour un cours doté de une à deux heures par semaine sur une année)
2. Sujets choisis dans le domaine des graphiques avec la tortue et le chapitre sur la robotique (en guise d'introduction à la science informatique ou pour des ateliers d'informatique si des robots LEGO EV3 ou NXT sont à disposition. Cela nécessite au moins 10-20 leçons).
3. Sujets choisis dans le domaine des graphiques avec la tortue suivis de la programmation de jeux (une à deux heures par semaine sur une année au degré secondaire supérieur).
4. Sujets choisis dans le domaine des graphiques avec la tortue, graphiques bidimensionnels et applications à des sujets scolaires (en guise d'introduction à l'informatique avec applications interdisciplinaires, 1 à 2 heures par semaine sur une année).
5. Sujets choisis dans le domaine des graphiques avec la tortue (en guise d'introduction à la programmation dans les cours d'informatique de base (Cours ICT), 4 à 10 heures.

Nous avons décidé de nommer les identifiant en anglais dans les programmes. Cela facilite non seulement la traduction du cours dans d'autres langues mais converge également vers une tendance à l'internationalisation des codes sources dans le monde du développement logiciel. En guise de soutien aux enseignants, chaque sujet abordé est accompagné d'une liste de mots-clés associés aux concepts fondamentaux en informatique introduits.

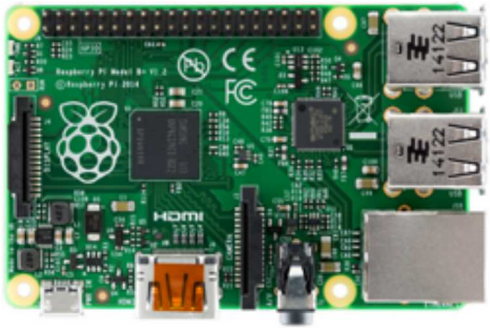
Solutions des exercices:

Les personnes travaillant pour une institution éducative peuvent demander les solutions des exercices par écrit à l'adresse ***help@tigerjython.com***. La demande doit inclure les informations suivantes qui doivent être vérifiables : nom, adresse, nom de l'institution, adresse de courriel. Les solutions sont à dispositions à condition de ne les utiliser qu'à titre privé et à ne les diffuser sous aucune forme.

Droits:

1.4 RASPBERRY PI

INTRODUCTION



On peut utiliser TigerJython sur le Raspberry Pi pour apprendre le langage de programmation Python pour accéder à ses ports GPIO ou sa carte son. Bien que TigerJython s'exécute de manière assez lente sur le Raspberry Pi, il offre une interface bien plus agréable que l'éditeur IDLE fourni avec Python. TigerJython met en effet à disposition les fonctionnalités suivantes fortes utiles : débogage, visualisation de l'état de la mémoire durant l'exécution du programme, graphiques avec la tortue, robotique, développement de jeux, etc …

INSTALLATION

La meilleure façon de débuter consiste à télécharger le système d'exploitation NOOS disponible sur **http://www.raspberrypi.org/downloads** et à le copier sur une carte SD de 8 Go au minimum et de classe 10 de préférence. Lors du démarrage, choisir le système *raspbian (*une variante de Linux Ubuntu optimisée pour le Raspberry Pi). Puisque cette distribution contient déjà une JRE, il suffit de copier le fichier *tigerjython2.jar* dans le dossier de votre choix, par exemple */home/pi/tigerjyton* et d'attribuer à ce fichier les droits d'exécution avec la commande *chmod* ou dans le gestionnaire de fichiers sous *File Properties*.

Pour démarrer TigerJython, saisir les commandes suivantes dans un terminal (console bash):

```
java -jar /home/pi/tigerjython/tigerjython2.jar
```

Pour être en mesure d'utiliser le module GPIO, TigerJython nécessite des droits administrateur. Il faut alors obligatoirement démarrer TigerJython en faisant précéder la commande d'invocation par le mot *sudo* :

```
sudo java -jar /home/pi/tigerjython/tigerjython2.jar
```

Au lieu de saisir à chaque fois cette commande, il est possible de spécifier dans le gestionnaire de fichiers que les fichiers dont l'extension est .jar doivent toujours être exécutés avec cette commande. Il est également possible de créer un script de démarrage dans le même but. Il est même possible de se simplifier encore un peu plus la vie en créant un raccourci sur le bureau de la manière suivante:

- Cliquer droit et copier l'icône IDLE puis le coller sur le bureau, afin de créer un nouvel icône de lien.

- Pour éditer le script de lancement associé à cet icône, cliquer droit sur l'icône et choisir *Leafpad.* Il est ensuite possible d'ajuster les entrées de manière appropriée et même de spécifier un icône représentant TigerJython (**téléchargement**). Voici un exemple:

```
[Desktop Entry]
Name=Tiger
Exec=sudo java -jar /home/pi/tigerjython/tigerjython2.jar
Icon=/usr/share/pixmaps/tjlogo1.png
```

Une fois ces manipulations effectuées, il est possible de démarrer Tiger Jython en cliquant sur l'icône. Il faudra se munir de patience (environ 1 minute) avant l'ouverture de Tiger Jython car le Raspberry Pi n'est pas une bête de course … Même si le chargement initial est lent, l'exécution des programmes Python est ensuite étonnamment rapide. *Remarque* : pour accélérer le chargement de TigerJython, prévoir une carte SD rapide (classe 10) et préférer un Raspberry Pi 2 nettement plus rapide que la première génération.

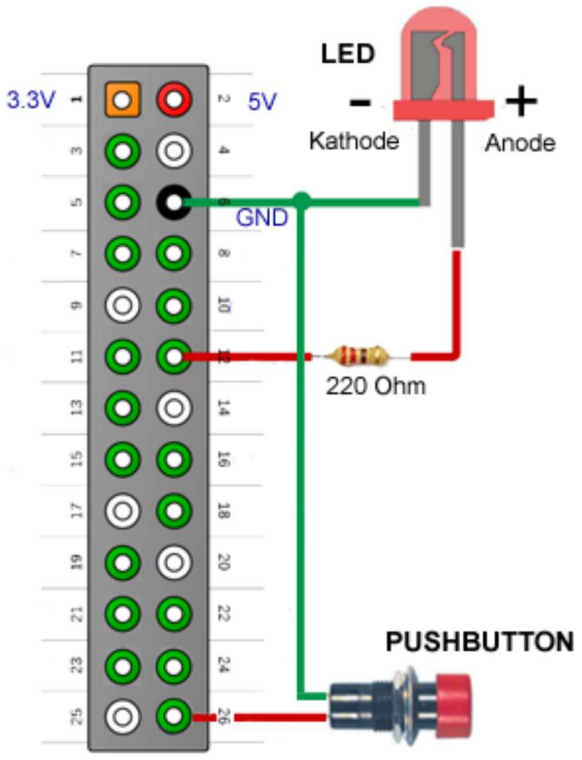
ENTRÉES / SORITES DIGITALES SUR LE GPIO

Le Raspberry Pi met à disposition 17 ports d'entrées / sorties digitales pouvant être branchés sur un connecteur 26 pins (40 pins pour la nouvelle version). Chaque canal peut être défini comme une entrée ou une sortie avec une résistance interne pull-up, pull-down ou sans résistance interne. Il y a des pins 5 V, 3.3 V et masse (GND) sur les connecteurs. **La tension d'entrée ne doit jamais excéder 3.3V, ce qui interdit le branchement de circuits externes dont les sorties sont en 5V directement sur les entrées GPIO.**

Dans TigerJython, on trouve la classe GPIO dans le module PRi_GPIO permettant de contrôler facilement les ports GPIO. Le module utilise la bibliothèque Pi4J de Robert Savage mais a l'avantage d'adopter une interface similaire au module **RPi.GPIO** disponible en standard pour l'implémentation CPython. Tous les fichiers nécessaires sont inclus dans la distribution de TigerJython.

Par défaut, les pins du port GPIO sont utilisés avec les nombres 1 à 26. Chaque pin peut être configuré avec *GPIO.setup()* en tant que canal d'entrée ou de sortie. On peut assigner une valeur à un canal donné avec *GPIO.output(channel, state)*. De manière similaire, on peut lire la valeur présente en entrée avec *GPIO.input(channel)*. Une documentation détaillée est disponible dans le menu d'aide sous *Aide > Documentation > APLU.*

Voici un montage permettant de tester les possibilités des ports GPIO. On y branche en série une LED et une résistance de 220 Ohms entre le port 6 (masse) et le port 12. Il faut remarquer qu'il faut disposer la LED dans le bon sens et branchant la cathode (patte la plus courte) sur la masse et l'anode (patte longue) sur le port 12. En cas de doute, il n'y a pas de souci à se faire, il faut juste la tourner si rien ne se passe. On connecte encore un boutton-poussoir entre le port 6 et le port 26.



Le programme suivant commence par définir le canal 12 comme un port de sortie et fait ensuite clignoter la LED 10 fois par seconde en attendant 100 ms entre chaque inversion du signal.

```
from RPi_GPIO import GPIO

GPIO.setup(12, GPIO.OUT)

while True:
    GPIO.output(12, 1)
    GPIO.delay(100)
    GPIO.output(12, 0)
    GPIO.delay(100)
```

Pour utiliser un port d'entrée, on peut connecter un boutton-poussoir au port 26. Cela nécessite typiquement quelques drapeaux permettant d'enclencher et déclencher le clignotement.

```
from RPi_GPIO import GPIO

# pin numbers
switch = 26
led = 12

print "Press button turn blinking on/off"

GPIO.setup(led, GPIO.OUT)
GPIO.setup(switch, GPIO.IN, GPIO.PUD_UP)
buttonPressed = False
blinking = False
ledOn = False

while True:
    v = GPIO.input(switch)
    if not buttonPressed and v == GPIO.LOW:
        buttonPressed = True
        blinking = not blinking

    if buttonPressed and v == GPIO.HIGH:
        buttonPressed = False

    if blinking:
        if ledOn:
            ledOn = False
            GPIO.output(led, GPIO.LOW)
        else:
            ledOn = True
            GPIO.output(led, GPIO.HIGH)
    else:
        ledOn = False
        GPIO.output(led, GPIO.LOW)
    GPIO.delay(100)
```

Sélectionner tout le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Dans la boucle, on commence par lire l'état actuel du boutton-poussoir. On nomme « polling » la technique consistant à lire de façon répétée la valeur d'un port. Lorsque ce dernier est enfoncé, il connecte la masse (port 6) au port 26, ce qui a pour conséquence de mettre le port 26 en tension basse (0V). L'instruction *input(26)* retourne donc la valeur GPIO LOW qui vaut 0. Lorsque le boutton-poussoir est relâché, la résistance interne pull-up du port 26 force la tension sur HIGH (3.3V) sans qu'il soit nécessaire d'appliquer cette tension depuis l'extérieur.

L'utilisation d'un boutton-poussoir est un peu compliquée du fait que, malgré le polling continuel du port 26, il est nécessaire de détecter un changement de tension en générant un événement qui ne survient que lors d'un passage d'une tension haute vers une tension basse. On utilise pour cela le drapeau *buttonPressed* qui passe à *True* lors d'une première pression sur le bouton. Suite à cela, on ne repasse pas dans la partie du code correspondante jusqu'à ce que le bouton soit pressé à nouveau après avoir été relâché.

Puisque la boucle est également responsable du clignotement, on utilise un drapeau *blinking* qui représente le statut allumé/éteint du clignotement. Le drapeau *lenOn* permet de se rappeler dans chaque itération si la LED doit être allumée ou éteinte.
En fin de boucle, puisque l'on ne sait pas si la LED est allumée ou éteinte lorsque le bouton pressoir est relâché, on met le pin 12 sur LOW pour être certain qu'elle soit éteinte. Le fait que le programme entre ensuite de manière répétée et de manière inutile dans le dernier *else* est une petite imperfection dont on peut s'accommoder.

Un ingénieur en électronique sait bien que lorsque le bouton est pressé, le contact produit des

rebonds (*bounces* en anglais) avant de se stabiliser. Cela n'est pas gênant dans notre situation puisque le signal a largement le temps de se stabiliser en 100ms.

UTILISER LES ÉVÉNEMENTS

Au lieu d'utiliser de nombreux drapeaux dans la boucle, il vaut mieux utiliser la programmation événementielle. Dans ce modèle de programmation, le fait d'enfoncer et de relâcher le boutton-poussoir correspond à un événement qui va automatiquement appeler la fonction *onButtonPressed()* appelée ***fonction de rappel***. Il suffit alors de maintenir le drapeau *blinking*.

```
from RPi_GPIO import GPIO

def onButtonPressed(channel, state):
   global blinking
   blinking = not blinking

# pin numbers
switch = 26
led = 12

print "Press button to turn blinking on/off"
GPIO.setup(led, GPIO.OUT)
GPIO.setup(switch, GPIO.IN, GPIO.PUD_UP) # pull-up resistor
GPIO.add_event_detect(switch, GPIO.FALLING) # event on falling edge
GPIO.add_event_callback(switch, onButtonPressed) # register callback

blinking = False
while True:
    if blinking:
        GPIO.output(led, 1)
        GPIO.delay(100)
        GPIO.output(led, 0)
    GPIO.delay(100)
```

Sélectionner tout le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Pour utiliser les événements, il suffit de spécifier à l'aide de la méthode *add_event_detect()* si l'on désire déclencher l'événement sur un front montant (transition de la tension basse à la tension haute), un front descendant (transition de la tension haute vers la tension basse) ou les deux. Suite à cela, on spécifie à l'aide de *add_event_callback()* une fonction de rappel qui va être appelée à chaque fois que l'événement de pression survient.

chapitre deux

TORTUE GRAPHIQUE

Objectifs d'apprentissage

- Être capable d'écrire un programme simple et de dessiner des figures à l'écran avec la tortue.
- Être capable de changer la couleur des lignes et des surfaces tracées par la tortue ainsi que d'ajuster la largeur des lignes.
- Savoir comment indiquer à la tortue de répéter plusieurs fois les mêmes instructions.
- Savoir comment exécuter une partie du programme uniquement sous certaines conditions.
- Être capable de définir ses propres commandes avec des paramètres donnés.
- Savoir ce que sont les variables et comment les utiliser dans ses programmes.
- Savoir ce qu'est la récursion et être capable d'écrire de petits programmes récursifs.
- Être capable de créer des objets-tortues et commander simultanément plusieurs tortues.

"A turtle is located at a certain place and it also has a certain viewing direction. Therefore, a turtle is like a person... children can identify with the turtle and can transfer their knowledge of their bodies into the learning of geometry."

Seymour Papert

2.1 DÉPLACER UNE TORTUE

INTRODUCTION

Programmer consiste à donner des ordres à une machine afin de la contrôler. La première machine que tu vas contrôler est une petite tortue se déplaçant à l'écran : on l'appellera tout simplement *la tortue*. Tu dois te demander ce dont cette fameuse tortue est capable et comment tu vas bien pouvoir la contrôler?

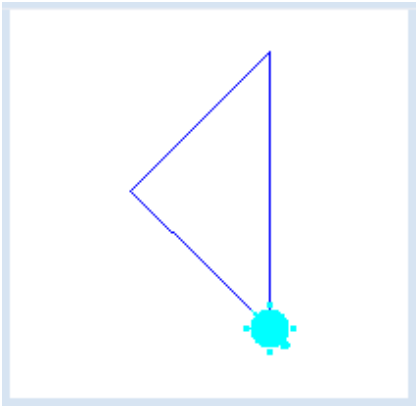
La tortue parle un langage de programmation génial qui s'appelle *Python* et qui est abondamment utilisé par les plus grands noms de l'informatique comme Google ou Dropbox. Pour le moment, disons qu'elle comprend des instructions anglaises suivies d'une paire de parenthèses entre lesquelles on peut donner des indications supplémentaires sur la manière d'exécuter l'instruction. Si aucune indication n'est nécessaire, il faut tout de même veiller à placer la paire de parenthèse vide. Fais bien attention à la différence entre les majuscules et les minuscules lorsque tu lui transmets tes instructions: notre tortue est de nature très pointilleuse!

La tortue peut se déplacer à l'intérieur d'une fenêtre et laisse une trace colorée sur son passage. Avant de pouvoir utiliser la tortue, il faut cependant instruire l'ordinateur sur la manière de la créer. C'est le rôle de l'instruction *makeTurtle()*. Pour faire bouger la tortue, il faut utiliser les instructions *forward(distance), left(angle)et right(angle)*.

CONCEPTS DE PROGRAMMATION: *éditer un code source, exécuter un programme, séquence d'instructions.*

TON PREMIER PROGRAMME

Voici le résultat de l'exécution de ton premier programme par la tortue. Clique sur « sélectionner le code source » et insère le code source dans l'éditeur de TigerJython. Tu peux l'exécuter en cliquant sur le bouton « start ». La tortue va immédiatement et de la manière la plus docile du monde tracer le triangle rectangle représenté ci-contre.



Les instructions comprises par la tortue sont définies dans un fichier externe gturtle nommé **module** dans le jargon de Python.

L'instruction **import** précise que l'on doit charger ces instructions avant de pouvoir les utiliser. L'instruction **makeTurtle()** créer une fenêtre contenant une tortue que ton programme va contrôler. Les lignes **lignes suivantes** présentent les instructions exécutées par la tortue à proprement parler.

```
from gturtle import *

makeTurtle()

forward(141)
left(135)
forward(100)
left(90)
forward(100)
```

MEMENTO

Au début de chaque programme, il est impératif de charger le module turtle et de créer une nouvelle tortue:

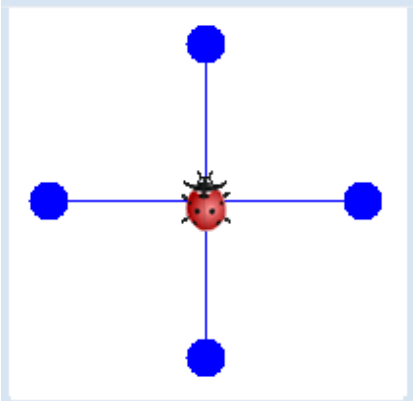
```
from gturtle import *
makeTurtle()
```

Ensuite seulement, tu peux passer un nombre quelconque d'instructions à la tortue. Elle reconnaît au moins les instructions suivantes (tu verras plus tard qu'elle en comprend encore bien d'autres):

forward(s)	Advancer de s (en pixels).
left(w)	Tourne la tortue d'un angle alpha (en degrés) vers la gauche.
right(w)	Tourne la tortue d'un angle alpha (en degrés) vers la droite.

PERSONNALISER TA TORTUE

Il est possible de passer un argument à la fonction **makeTurtle()** permettant d'indiquer l'image à utiliser en guise de tortue pour donner une petite touche personnelle à ton programme. Ici, on utilise par exemple l'image beetle.gif du dossier sprites de la distribution de TigerJython. Remarque bien qu'il faut indiquer le chemin vers le fichier entre double-guillemets "chemin/vers/fichier".



```
from gturtle import *

makeTurtle("sprites/beetle.gif")

forward(100)
dot(20)
back(100)
right(90)

forward(100)
dot(20)
back(100)
right(90)

forward(100)
dot(20)
back(100)
right(90)

forward(100)
dot(20)
back(100)
right(90)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

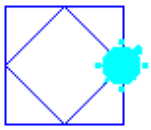
MEMENTO

Si tu désires utiliser une autre image pour représenter ta tortue, il faut en créer une à l'aide d'un éditeur d'images. Pour que tout fonctionne au mieux, il faut utiliser une image au format png ou gif qui a une taille de 32x32 pixels et un arrière-fond transparent. Il faut ensuite déposer l'image à utiliser dans un dossier nommé sprites dans le même dossier que ton programme Python.

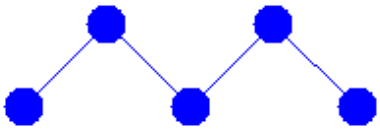
Dans ce programme, tu utilises la nouvelle instruction **back()** permettant de faire faire un demi-tour à la tortue ainsi que **dot(r)** qui permet de dessiner un disque plein de rayon r.

EXERCICES

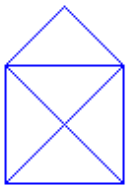
1. Avec la tortue, dessine deux carrés emboîtés semblables à la figure ci-contre.



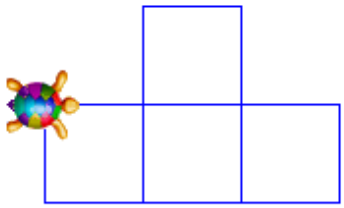
2. Utilise l'instruction dot() pour générer la figure ci-contre



3. Voici un jeu pour enfants nommé « La maison de Nicolas ». Le but est de dessiner la maison ci-contre sans lever le crayon avec exactement 8 segments droits sans repasser deux fois sur un même segment. Dessiner la maison de Nicolas avec la tortue.



4*. Créer une image personnelle pour ta tortue à l'aide d'un éditeur d'image et utilise ta tortue personnalisée pour générer l'image ci-contre. La longueur du côté d'un carré est de 100 et tu commencer et terminer le dessin où bon te semble.



2.2 UTILISER DES COULEURS

INTRODUCTION

La tortue dessine sa trace en utilisant un crayon de couleur qu'il est possible de personnaliser avec certaines instructions spécifiques. Aussi longtemps que le crayon est posé sur la feuille (*down*), la tortue dessine une trace. Il est possible d'interrompre ce comportement avec l'instruction penUp() qui va lever le crayon de la feuille et empêcher la tortue de dessiner. Il est ensuite possible de reposer le crayon sur la feuille avec penDown() de sorte qu'elle se remette à dessiner la trace.

L'instruction setPenColor(color) permet de changer la couleur du crayon. Il est capital de placer le nom de la couleur en anglais et entre doubles guillemets. Voici une liste partielle de couleurs utilisables : *yellow, gold, orange, red, maroon, violet, magenta, purple, navy, blue, skyblue, cyan, turquoise, lightgreen, green, darkgreen, chocolate, brown, black, gray, white.*

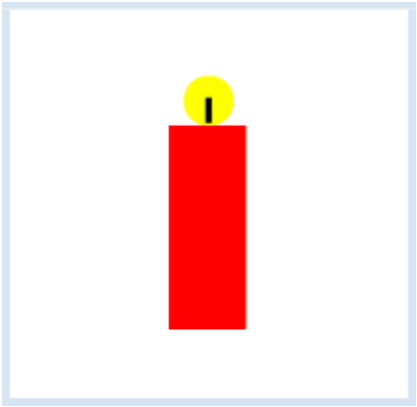
CONCEPTS DE PROGRAMMATION: *dessiner avec des couleurs*

COULEUR ET LARGEUR DU CRAYON

Ce programme ordonne à la tortue de dessiner une bougie avec une ligne **rouge** très large. On règle la largeur de la trace en pixels avec instruction **setLineWidth()**.

Pour dessiner la flamme **jaune**, il suffit d'utiliser dot(). Il y a une partie du trajet durant laquelle la tortue ne dessine plus de trace parce que le crayon a été levé avec **penUp()**. Dès qu'elle voit l'instruction **penDown()** elle se remet à dessiner.

hideTurtle() permet de rendre la tortue invisible.



```
from gturtle import *

makeTurtle()

setLineWidth(60)
setPenColor("red")
forward(100)
penUp()
forward(50)
penDown()
setPenColor("yellow")
dot(40)
setLineWidth(5)
setPenColor("black")
back(15)
hideTurtle()
```

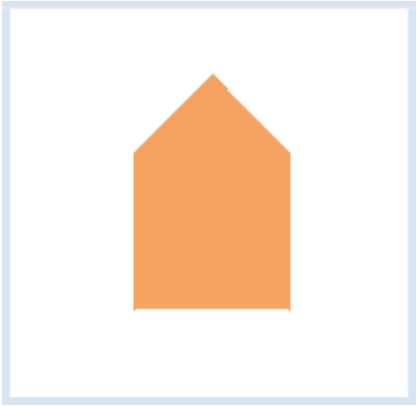
Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

On peut changer la couleur du crayon avec l'instruction **setPenColor(color)** où color est le nom d'une couleur en anglais. L'instruction **penUp()** demande à la tortue d'arrêter de dessiner et **penDown()** lui demande de recommencer à dessiner. Tu peux controller la largeur du trait avec **setLineWidth(width)** où widthest la largeur en pixels.

La tortue connaît les couleurs ***X11***. Il en existe plusieurs dizaines que tu peux trouver sur le Web **http://cng.seas.rochester.edu/CNG/docs/x11color.html** . Tu peux toutes les utiliser avec l'instruction *setPenColor(color)*.

SURFACES PLEINES



La tortue peut remplir toute surface qu'elle a délimitée avec son crayon d'une certaine couleur. L'instruction **startPath()**, permet d'indiquer que tu veux remplir la surface délimitée par la trace dessinée. La tortue mémorise alors sa position actuelle comme le début d'une série de segments. À partir de ce moment, tu peux déplacer la tortue librement sur la surface. Ensuite, dès que l'instruction **fillPath()**, est rencontrée, le point où se trouve la tortue est relié avec le point de départ mémorisé précédemment et la surface ainsi délimitée est remplie de couleur. La couleur de remplissage peut être spécifiée avec l'instruction **setFillColor(color)**.

Les lignes du programme débutant par un dièse (#) sont appelées des **commentaires**, et sont ignorées lors de l'exécution du programme. Il est utile et même fortement recommandé d'ajouter des commentaires au programme pour le rendre plus clair ou te permettre de comprendre un code compliqué quelques semaines plus tard.

```
from gturtle import *

makeTurtle()

setPenColor("sandybrown")
setFillColor("sandybrown")
startPath()
forward(100)
right(45)
forward(72)
right(90)
forward(72)
right(45)
forward(100)
fillPath()
hideTurtle()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

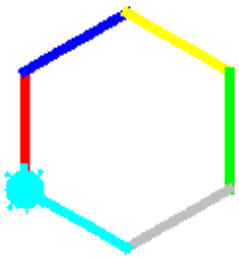
MEMENTO

Si tu veux remplir une surface délimitée par une série de lignes, tu commences par appeler **startPath()** avant de dessiner. Ultérieurement, l'instruction **fillPath()** permet de relier le point de départ du tracé avec la position courante de la tortue et de remplir la surface fermée ainsi délimitée.

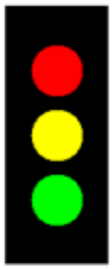
Tu peux (et devrais) agrémenter ton code de **commentaires** pour le rendre plus compréhensible par ceux qui te liront et pour toi-même trois semaines plus tard lorsque tu auras bien oublié ce que signifie ton code. Les commentaires commencent par un dièse (#).

EXERCICES

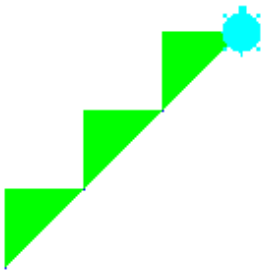
1. Dessiner un hexagone régulier avec la tortue et assure-toi que chaque côté a une couleur différente



2. Dessiner un feu de circulation. Tu peux dessiner le rectangle noir avec un crayon de largeur 80 et les cercles avec *dot(40)*.



3. Dessiner l'image ci-contre avec la tortue.



2.3 RÉPÉTITION

INTRODUCTION

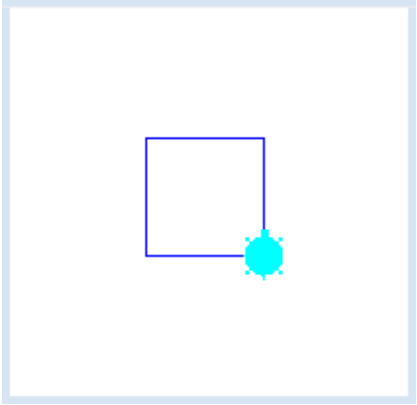
Les ordinateurs sont particulièrement bons pour répéter des instructions de manière répétitives des milliers de fois, y compris les instructions à destination de la tortue. Pour dessiner un carré, il faut entrer les commandes *forward(100)* et *left(90)* quatre fois d'affilée. Pour ce faire, il est suffisant d'indiquer à la tortue de répéter quatre fois ces deux instructions. Cela se fait facilement avec la structure *repeat* qui permet d'indiquer à la tortue de répéter un bloc d'instructions un certain nombre de fois. Afin d'indiquer à l'ordinateur quelles instructions il faut répéter, ces dernières doivent être décalées (on dit *indentées* en langage technique) par rapport à l'instruction *repeat*. On utilise généralement 4 espaces consécutifs pour marquer une indentation.

CONCEPTS DE PROGRAMMATION: *Répétition simple, boucle de répétition au lieu de copier-coller inutilement du code.*

STRUCTURE REPEAT

Pour dessiner un carrée, la tortue doit avancer tout droit 4 fois de suite et tourner à chaque fois de 90°. Si l'on devait écrire un tel programme, il deviendrait rapidement très long.

À l'aide de l'instruction **repeat 4:** on demande à la tortue de répéter 4 fois les **Lignes indentées**. Faites attention de ne pas oublier le double point (:) après la structure *repeat*.



```
from gturtle import *

makeTurtle()
repeat 4:
    forward(100)
    left(90)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

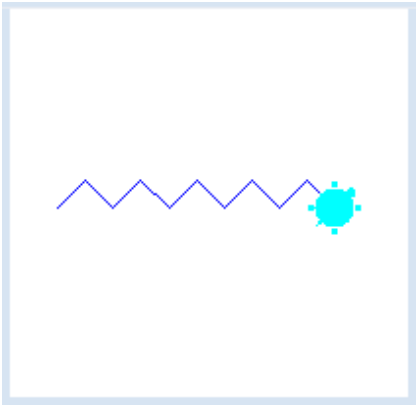
MEMENTO

Par l'instruction *repeat n:*, on demande à l'ordinateur de répéter *n* fois une ou plusieurs instructions avant de continuer l'exécution normale du programme. Tout ce qui doit être répété doit être placé directement sous l'instruction *repeat* avec une indentation cohérente. Il ne faut pas mélanger les espaces et les tabulations pour marquer l'indentation. Dans le monde de Python, les espaces sont recommandés pour marquer les tabulations.

```
repeat number:
    Instructions
    qui doivent être
    répétées
```

RÉPÉTER DES SONS

Un exemple typique de répétition de sons est le "Pim-Pom Pim-Pom" des camions pompiers. Il est facile de générer ce genre de suite de sons et, pour le plaisir, faire zizaguer la tortue. On peut générer un son pur avec l'instruction **playTone()**, en indiquant sa hauteur grâce à la fréquence (en Hertz) et sa durée (en millisecondes).



```
from gturtle import *

makeTurtle()

setPos(-200, 0)
right(45)

repeat 5:
    playTone(392, 400)
    forward(50)
    right(90)
    playTone(523, 400)
    forward(50)
    left(90)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

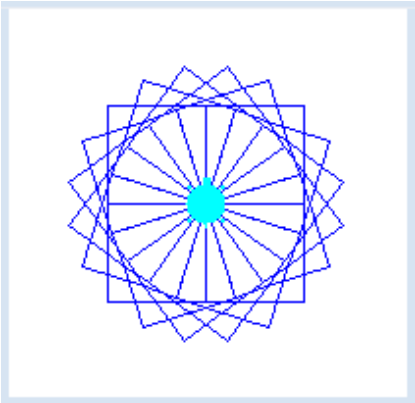
La fonction **setPos(x, y)** permet de placer la tortue à un endroit précis sur la fenêtre sans dessiner de trace.

Les deux entiers *x* et *y* représentent les coordonnées relative à l'origine qui se trouve au centre de la fenêtre. Les intervalles dans lesquels se situent les coordonnées dépendent de la taille de la fenêtre

Tu peux également spécifier la note de la gamme du son produit par **playTone()** en utilisant une lettre représentant la note à jouer dans le système anglo-saxon. Pour rappel, le *La* est représenté par la lettre *a*, le *Si* par la lettre *b* etc ... Pour passer à l'octave inférieure, il faut rajouter une apostrophe après la lettre de la note, ce qui donnerait a', b', c', d', et ainsi de suite. On peut rajouter deux apostrophes pour passer à l'octave supérieure. Il ne faut pas oublier de rajouter les doubles guillemets autour du nom de la note, donc par exemple *"a"*. On peut spécifier un instrument pour jouer cette note. Les instruments disponibles sont "piano", guitar, "harp", "trumpet", "xylophone", "organ", "violin", "panflute", "bird", "seashore"). Faites-vous plaisir en essayant ceci :

Son grave: *playTone("g", 400, instrument = "trumpet")*
Son aigu: *playTone("c", 400, instrument = "trumpet")*

BOUCLES DE RÉPÉTITIONS IMBRIQUÉES



On peut facilement générer un carré avec *repeat 4:*. Maintenant, on va essayer de dessiner 20 carrés en les tournant légèrement les uns par rapport aux autres. Pour ce faire, il faut imbriquer deux instructions *repeat* l'une dans l'autre. Dans le **bloc d'instructions interne**, la tortue dessine un carré et tourne ensuite de 18 degrés vers la droite. L'instruction **repeat 20** permet de répéter ce comportement 20 fois. Faites bien attention à indenter correctement les différents blocs d'instructions, sans quoi la tortue risque de faire n'importe quoi.

On peut accélérer énormément le dessin en cachant la tortue avec **hideTurtle()**.

```
from gturtle import *

makeTurtle()

# hideTurtle()
repeat 20:
    repeat 4:
        forward(80)
        left(90)
    right(18)
```

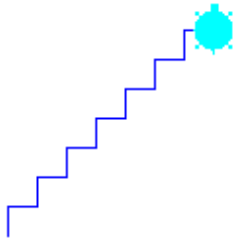
Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

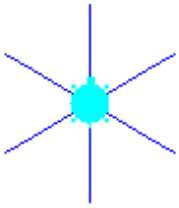
Les instructions *repeat* peuvent être imbriquées. Il est capital d'indenter correctement les lignes appartenant à chacun des blocs *repeat*.

EXERCICES

1. Dessiner un escalier comportant 7 marches.

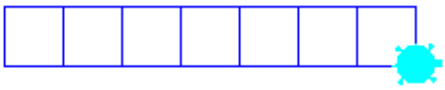


2. Dessiner une étoile en utilisant l'instruction *back()*.

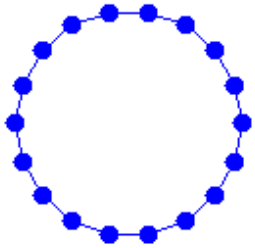


3. Dessiner une "vraie" étoile grâce en utilisant les angles de 140° et 80°.

4. Dessiner la figure suivante en utilisant deux instructions *repeat* imbriquées. Le *repeat* intérieur doit dessiner un carré.



5. Dessiner un collier de perles.



6. Dessiner un oiseau.

2.4 FONCTIONS

INTRODUCTION

Lors de la réalisation d'une grande image, on veut utiliser à de nombreuses reprises plusieurs figures telles que des triangles ou des carrés. Le souci est que la tortue ne sait pas dessiner des triangles ou des carrés, elle ne sait que tracer des lignes. De ce fait, il est nécessaire d'expliquer à la tortue comment dessiner telle ou telle figure à l'aide de plusieurs instructions, ce qui donne vite lieu à du code répétitif si l'on veut dessiner plusieurs figures de même type. N'y aurait-il pas une solution plus facile pour réaliser ces figures?

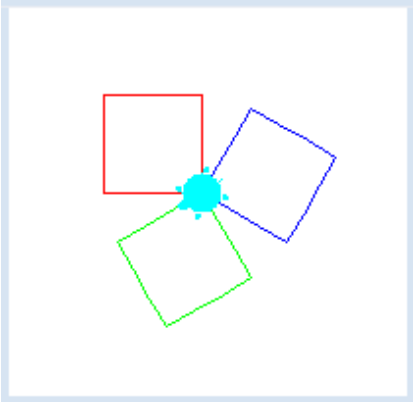
Bien sûr que oui! On peut enseigner à la tortue de nouvelles commandes, comme le dessin de carrés ou de triangles. Il suffit ensuite de lui demander de dessiner un carré ou un triangle au lieu de détailler chaque instruction individuelle. Pour ce faire, il faut choisir un nom d'identifiant valide, par exemple *square*, et écrire *def square():* (remarquer le double point qui fait partie de la syntaxe). Ensuite, il suffit d'écrire toutes les instructions individuelles qui constitueront la nouvelle commande. Pour permettre à l'interpréteur de distinguer les commandes qui font partie de la fonction nouvellement définie et ne pas les confondre avec les instructions « indépendantes », il faut indenter toutes les instructions prenant part au bloc définissant la nouvelle fonction.

CONCEPTS DE PROGRAMMATION: *Programmation modulaire, définition/appel de fonction*

DÉFINIR SES PROPRES COMMANDES

Dans ce programme, on utilise le mot-clé ***def*** pour définir une nouvelle fonction **square()**. Die Turtle weiss danach, wie sie ein Quadrat zeichnen kann, sie hat aber noch keines gezeichnet.

Celle-ci va permettre à la tortue de savoir comment dessiner des carrés mais ne va pas encore lui dire d'en dessiner un. Avec l'instruction *square()*, la tortue va dessiner un carré **carré** de côté 100 à sa position actuelle. Dans notre exemple, on dessine un carré rouge, un bleu et un vert.



```
from gturtle import *

def square():
    repeat 4:
        forward(100)
        left(90)

makeTurtle()
setPenColor("red")
square()
right(120)
setPenColor("blue")
square()
right(120)
setPenColor("green")
square()
```

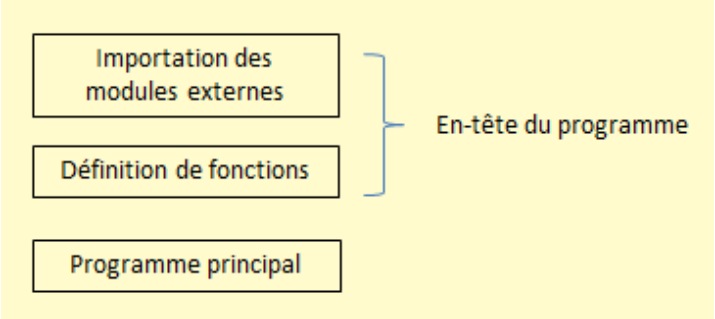
MEMENTO

Pour définir une nouvelle commande, on utilise la syntaxe ***def identifiant():*** Il est conseillé d'utiliser un nom pertinent qui reflète bien ce que fait la commande. Toutes les lignes d'instructions prenant part au bloc définissant la nouvelle commande doivent être indentées de manière cohérente.

```
def identifier():
    instructions
```

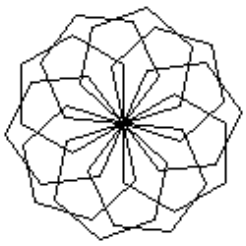
Il ne faut pas oublier de placer des parenthèses et le double-point après le nom de la commande ! Dans la terminologie Python, on appelle les commandes définies par le programmeur des *fonctions*. Lorsqu'on utilise la fonction *square*(), on dit en termes techniques qu'elle est « *appelée* ».

Il faut prendre l'habitude de définir les nouvelles fonctions dans l'en-tête du programme puisqu'elles doivent être définies avant d'être appelées.

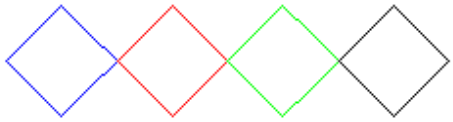


EXERCICES

1. Définir une commande *hexagon()* permettant de dessiner un hexagone. Utiliser ensuite cette nouvelle fonction pour dessiner la figure ci-contre.



2a. Définir une fonction permettant de dessiner un carré posé sur l'un de ses sommets. Utiliser ensuite cette fonction pour dessiner la figure ci-contre.

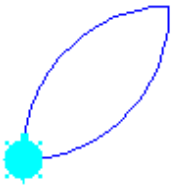


2b*. Pour dessiner des formes pleines, on peut utiliser les fonctions *startPath()* et *fillPath()*.

3a. Dans les exercices 3a à 3e, on va découvrir comment résoudre un problème complexe par étapes en utilisant des fonctions. Il s'agit d'un concept essentiel en programmation nommé *programmation procédurale*.

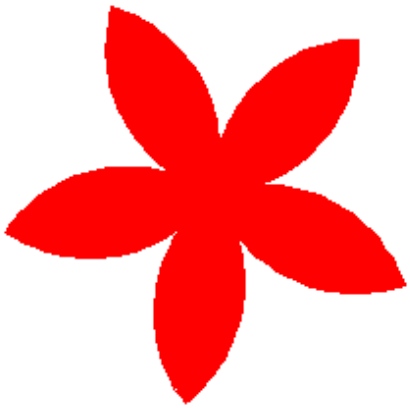
Définir une fonction *arc()* qui demande à la tortue de dessiner un arc de cercle d'angle au centre -90° (orienté vers la droite). Pour accélérer le dessin, on peut régler la vitesse de la tortue au maximum avec *speed(-1)*.

3b. Ajouter au programme la fonction *petal()* chargée de dessiner deux arcs de cercles consécutifs de sorte que la tortue se retrouve dans sa position et son orientation de départ à la fin du dessin.

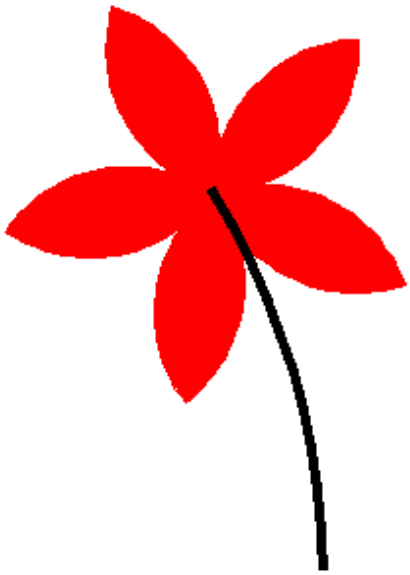


3c. Modifier le programme précédent pour de sorte que la fonction *petal()* dessine une forme remplie de rouge sans bord visible.

3d. Étendre le programme avec la fonction *flower()* permettant de dessiner une fleu à cinq pétales. Pour accélérer encore davantage le dessin, on peut utiliser la fonction *hideTurtle()* pour rendre la tortue invisible.



3e*. Ajouter une tige à la fleur en définissant une nouvelle fonction nommée de façon appropriée.



2.5 PARAMÈTRES

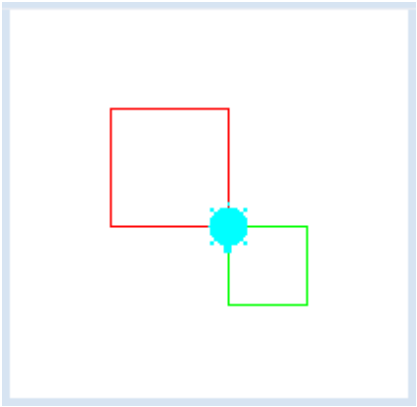
INTRODUCTION

Lorsque l'on utilise la commande *forward()*, on indique à la tortue de combien on veut avancer en indiquant la valeur entre parenthèses. Cette valeur entre parenthèses indique donc de combien on veut faire avancer la tortue. Il s'agit d'une précision sur la manière d'exécuter la commande et s'appelle un *paramètre*. Dans le cas précis, il s'agit d'un nombre entier qui peut varier à chaque appel de la fonction *forward()*. Dans le chapitre précédent, nous avons défini notre propre commande *square()* mais celle-ci, au contraire de la fonction *forward()*, ne prend aucun paramètre et dessine inlassablement des carrés dont le coté vaut 100. Il serait bien pratique d'être en mesure de préciser la taille du côté du carré. Comment peut-on faire cela ?

CONCEPTS DE PROGRAMMATION: *Paramètres, passer des paramètres*

COMMANDES AVEC PARAMÈTRES

Le programme ci-dessous définit également une fonction *square* permettant de dessiner un carré. Cependant, au lieu d'utiliser des parenthèses vides dans la définition de la fonction, on ajoute le paramètre formel *sidelength* que l'on peut spécifier lors de l'appel **forward(sidelength)**. On peut de ce fait utiliser la fonction *square* à plusieurs reprises en spécifiant à chaque fois une autre valeur pour la longueur des côtés *sidelength*. Par exemple, l'appel **square(80)** demande à la tortue de dessiner un carré de 80 pixels de côté alors que l'appel **square(50)** lui demande d'en dessiner un de 50 pixels de côté.



```
from gturtle import *

def square(sidelength):
    repeat 4:
        forward(sidelength)
        left(90)

makeTurtle()
setPenColor("red")
square(80)
left(180)
setPenColor("green")
square(50)
```

MEMENTO

Les paramètres jouent le rôle de valeurs de substitution pouvant être différente à chaque fois. Lors de la définition d'une nouvelle commande, on spécifie entre parenthèses et dans l'ordre le nom des paramètres formels tels que l'on s'y réfère dans le corps de la fonction.

```
def commandname(parameter):
    Instructions that
    use  parameter
```

Les noms des paramètres formels peuvent être choisis librement mais ils devraient refléter leur rôle dans le corps de la fonction. Lors de l'appel de la fonction, on spécifie également entre parenthèses la valeur que l'on veut substituer au paramètre au sein de la fonction.

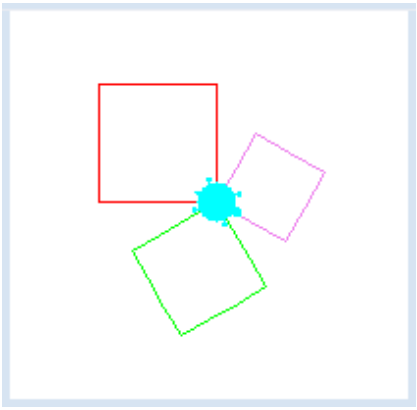
```
commandname(123)
```

En l'occurrence, la valeur du paramètre vaut 123. Toutes les occurrences du paramètre au sein du corps de la fonction vont donc être remplacées par cette valeur durant l'intégralité de l'appel.

PARAMÈTRES MULTIPLES

Les commandes peuvent prendre plusieurs paramètres. Pour un carré, on pourrait utiliser **def square(sidelength, color)** pour ajouter un deuxième paramètre permettant de spécifier la couleur dans laquelle il faut dessiner les côtés.

Cela rend possible une utilisation encore bien plus flexible de la fonction *square()*. En effet, **square(100, "red")**dessinera un carré rouge de 100 pixels de côté tandis que **square(80, "green")** dessine un carré vert de 80 pixels de côté.



```
from gturtle import *

def square(sidelength, color):
    setPenColor(color)
    repeat 4:
        forward(sidelength)
        left(90)

makeTurtle()
square(100, "red")
left(120)
square(80, "green")
left(120)
square(60, "violet")
```

MEMENTO

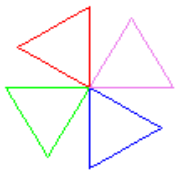
Les commandes peuvent accepter plusieurs paramètres. Ces derniers sont indiqués entre parenthèses et séparés par des virgules:

```
def commandname(param1, param2....):
   Instructions qui utilisent les paramètres param1 et param2
```

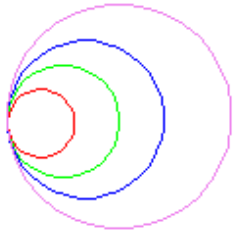
L'ordre des paramètres formels entre parenthèses dans la définition de la fonction doit correspondre à l'ordre des paramètres réels passés à la fonction lors de son appel.

EXERCICES

1. Définir une fonction *triangle(color)* qui demande à la tortue de dessiner des triangles colorés. Elle doit dessiner quatre triangles de couleur rouge, verte, bleue et violette.



2. Définir une fonction *colorCircle(radius, color)* telle que la tortue dessine un cercle coloré. La fonction *rightArc(radius, angle)* peut être utilisée dans ce but. Employer cette fonction par la suite pour dessiner la figure ci-contre.



3. Malheureusement, le programme suivant dessine trois pentagones isométriques au lieu de leur donner une taille différente comme souhaité. Expliquer le pourquoi du comment et réparer le programme.

```
from gturtle import *

def pentagon(sidelength, color):
    setPenColor(color)
    repeat 5:
        forward(90)
        left(72)

makeTurtle()
pentagon(100, "red")
left(120)
pentagon(80, "green")
left(120)
pentagon(60, "violet")
```

4. La fonction *segment(dist, angle)* permet de faire avancer la tortue sur un segment de longueur *dist* et de terminer son trajet en se tournant d'un angle *angle* :

```
def segment(s, w):
    forward(s)
    right(w)
```

Écrire un programme qui appelle cette fonction 92 fois avec *dist* = 300 et *angle* = 151. Débuter le dessin dans une position appropriée avec la fonction *setPos(x,y)*.

5*. Dans les exemples suivants, la tortue va effectuer des mouvements de deux, trois ou quatre segments. Admirer les graphiques produits dans les cas suivants :

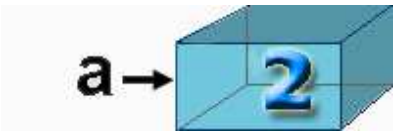
Nombre de segments	Valeurs	Nombre de répétitions
2	forward(77) right(140.86) forward(310) right(112)	37
3	forward(15.4) right(140.86) forward(62) right(112) forwad(57.2) right(130)	46
4	forward(31) right(141) forward(112) right(87.19) forward(115.2) right(130) forward(186) right(121.43)	68

2.6 VARIABLES

INTRODUCTION

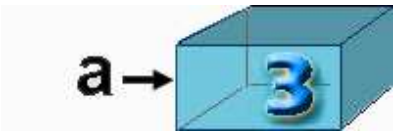
Dans le chapitre précèdent, nous avons dessiné des carrés dont la longueur était « codée en dur » dans le programme. Cela rend le programme très peu flexible car il faut le modifier à chaque fois que l'on veut dessiner des carrés de taille différente. Une solution serait de demander à l'utilisateur la taille des carrés à tracer à l'aide d'une boîte de dialogue. Cette solution nécessite cependant de stocker la valeur saisie par l'utilisateur quelque part dans la mémoire de l'ordinateur, dans une sorte de boîte que l'on appelle couramment **variable**. Une variable est donc une sorte de boîte contenue dans la mémoire RAM de l'ordinateur accessible par son nom. En bref, une variable possède un nom et une valeur. Les règles pour nommer les variables sont les même que pour les fonctions : il ne peut s'agir de mots réservés du langage Python (mots-clés) ni de caractères spéciaux. De plus, le nom d'une variable ne peut pas commencer par un chiffre.

L'instruction *a = 2* a pour effet de créer une boîte accessible par le nom *a* et y stocker la valeur *2*. Plus loin, nous verrons que l'on est ainsi en train de **définir** une variable *a* et de lui **affecter** la valeur *2*.



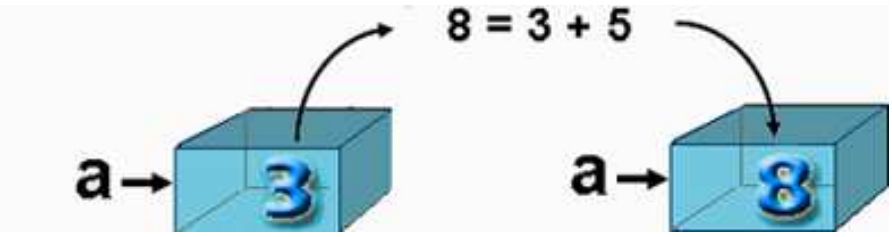
a = 2 :définition de variable (affectation)

Il n'y a de place que pour un seul objet dans la boîte. Ultérieurement, pour insérer la valeur *3* sous le nom *a*, il faut simplement écrire *a = 3* [**plus...**]



a = 3 : nouvelle affectation

Donc, que se passe-t-il lorsqu'on écrit *a = a + 5* ? Python évalue d'abord le côté droit de l'expression, *a + 5*, ajoutant ainsi *5* à la valeur actuelle contenue dans la variable *a*, et stocke ensuite le résultat de cette expression, à savoir le nombre entier *8* dans la variable indiquée à gauche du signe =, qui se trouve être la variable *a*.

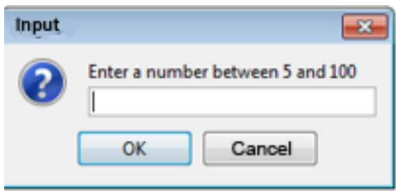


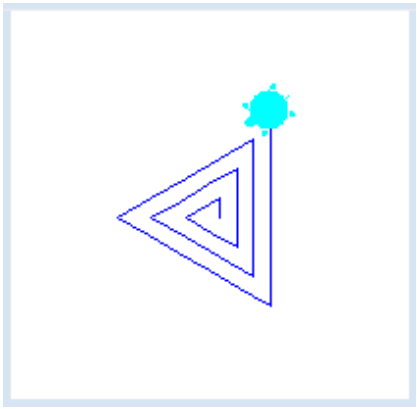
De ce fait, le signe d'égalité ne porte pas la même signification en programmation qu'en mathématiques. Il ne définit pas une égalité au sens d'une équation, mais plutôt la définition d'une variable ou l'affectation d'une valeur à une variable [**plus...**].

CONCEPTS DE PROGRAMMATION: *Définition de variables, affectations*

LIRE ET MODIFIER LE CONTENU D'UNE VARIABLE

La boîte de dialogue ci-dessous permet d'affecter une valeur entre 10 et 100 à la **Variablen x**. La boucle qui suit **modifie** la valeur de cette même variable pour dessiner une spirale.





```
from gturtle import *

makeTurtle()

x = inputInt("Enter a number between 5 and 100")
repeat 10:
    forward(x)
    left(120)
    x = x + 20
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

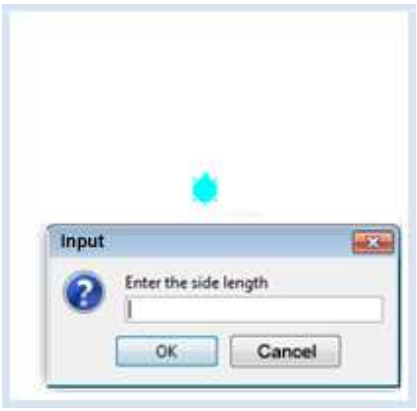
MEMENTO

Les ***variables*** permettent de stocker des valeurs qu'il est possible de lire et de modifier au cours de l'exécution du programme. Toute variable possède un nom et une valeur. On peut définir une variable et lui affecter une valeur à l'aide de symbole égal = [**plus...**].

DISTINCTION ENTRE VARIABLES ET PARAMÈTRES

Il est crucial de bien saisir la différence entre une variable et un paramètre. Les paramètres servent à "injecter" des données à l'intérieur d'une fonction et ne seront valide qu'à l'intérieur de cette fonction alors qu'il est possible d'utiliser des variables partout dans le programme. Lors de l'appel d'une fonction, on assigne une valeur à chacun des paramètres qui sont alors accessibles dans le corps de la fonction comme des variables. [**plus...**]

Pour bien expliciter la différence entre les deux, la fonction *square()* du programme ci-dessous utilise le paramètre **sidelength**. Lorsque l'on saisit un nombre avec **inputInt()**, celui-ci est stocké dans la **variable** *s*. Lors de l'appel de **square(),** on passe la valeur de la variable *s* au paramètre *sidelength* de la fonction *square()*.




```
from gturtle import *

def square(sidelength):
    repeat 4:
```

```
        forward(sidelength)
        right(90)

makeTurtle()
s = inputInt("Enter the side length")
square(s)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

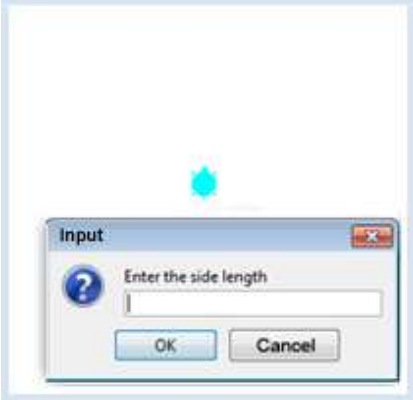
MEMENTO

Il faut bien distinguer entre la variable *s* et le paramètre *sidelength*. Dans la définition d'une fonction, les paramètres peuvent être considérés comme des valeurs de substitution et sont utilisés comme une variable dont l'existence est limitée à l'intérieur de la fonction.

Ainsi, si l'on appelle la fonction en passant une variable comme paramètre, c'est la valeur contenue dans la variable au moment de l'appel qui est utilisée en guise de paramètre lors de l'exécution de la fonction. De ce fait, *square(length)* dessine un carré dont la longueur du côté est la valeur contenue dans la variable *length* [**plus...**].

THE SAME NAME FOR DIFFERENT THINGS

Comme nous l'avons déjà vu, le nom donné aux paramètres et aux variables devraient refléter leur but et leur utilisation, mais ils peuvent être choisis librement. Il n'est de ce fait pas rare de choisir exactement le même nom pour une *variable* et le *paramèter* d'une fonction. Cela ne pose aucun conflit de nom. Il faut cependant bien distinguer les deux concepts pour être en mesure de comprendre correctement le programme.




```
from gturtle import *

def square(sidelength):
    repeat 4:
        forward(sidelength)
        right(90)

makeTurtle()
sidelength = inputInt("Enter the side length")
square(sidelength)
```

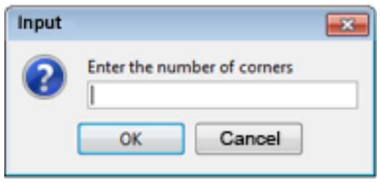
Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

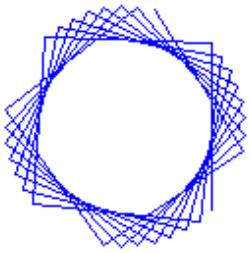
Bien qu'il soit possible d'utiliser le même nom pour un paramètre spécifique d'une fonction et une variable dans le programme sans que cela ne pose problème, il est essentiel de bien distinguer les deux sur le plan conceptuel.

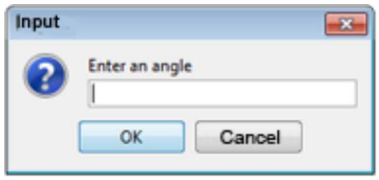
EXERCICES

1. Développer un programme qui demande à l'utilisateur un nombre entier *n* et qui dessine ensuite un polygone régulier à *n* côtés avec la tortue. Par exemple, si l'utilisateur saisit le nombre 8, il faut dessiner un polygone régulier à 8 côtés, à savoir un octogone. Le programme doit calculer l'angle de rotation approprié après chaque segment droit. Il peut être utile de se mettre dans la peau de la tortue et réfléchir de combien il faut tourner avant de dessiner le prochain segment droit. Pour information, nous avions déjà tracé des triangles équilatéraux qui sont des polygones réguliers à trois côtés.

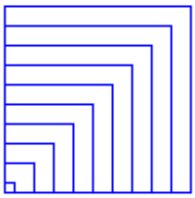


2. Développer un programme qui, après avoir demandé à l'utilisateur un angle par l'intermédiaire d'une boîte de dialogue, dessine 30 segments droits de longueur 100 et formant un angle correspondant les uns avec les autres. Tester ensuite le programme avec des angles différents pour essayer de dessiner des motifs sympathiques. Il est possible d'accélérer le dessin en cachant la tortue avec *hideTurtle()*.

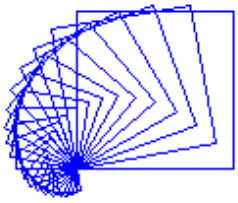




3. Demander à la tortue de dessiner 10 carrés. Définir tout d'abord une fonction *square* prenant le paramètre *side_length*. Le côté du premier carré mesure 8 et celui des carrés suivants est toujours supérieur de 10 à celui du carré précédent.



4. Lire le côté du plus grand carré depuis une boîte de dialogue. Dessiner ensuite 20 carrés dont le côté du carré suivant diminue toujours d'un **facteur** 0.9 par rapport au côté du carré précédent.



2.7 BRANCHEMENTS CONDITIONNELS (SÉLECTION)

INTRODUCTION

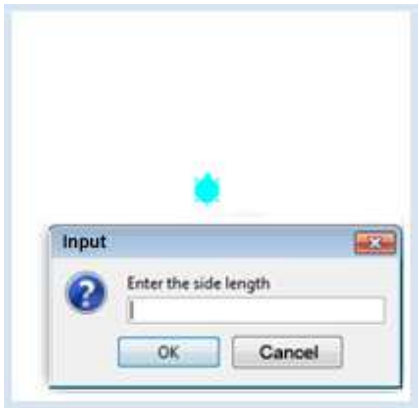
Ce que l'on entreprend dans la vie de tous les jours dépend souvent de plusieurs conditions. Par exemple, la manière dont vous allez vous rendre à l'école dépend de la météo. Vous vous dites : « S'il pleut aujourd'hui, je prends les transports publics et dans le cas contraire, j'y vais en vélo ». De manière semblable, le flux d'instructions dans un programme peut également dépendre de certaines conditions. Les ***branchements conditionnels*** sont des ***structures de contrôle*** permettant à un programme de modifier son exécution sur la base de certaines conditions et sont essentiels à n'importe quel langage de programmation. Les instructions suivant le *if*` sont uniquement exécutées si la condition est vraie et, dans le cas contraire, ce sont les instructions du bloc *else* qui sont exécutées.

CONCEPTS DE PROGRAMMATION: *Conditions, branchements conditionnels, sélection, structure if-else*

VALIDATION DE SAISIES

Après la **saisie de la longueur du côté** dans la boîte de dialogue, le carré n'est dessiné que s'il tient complètement dans les limites de la fenêtre.

Il faut donc examiner la valeur de la variable *s*. **Si s est inférieur à 300**, un carré de côté *s* est dessiné, **dans le cas contraire** (*else*), un message apparaît dans la partie inférieure de la fenêtre de TigerJython. En programmation, cette construction est réalisée à l'aide d'une structure *if*.




```
from gturtle import *

def square(sidelength):
    repeat 4:
        forward(sidelength)
        right(90)

makeTurtle()
s = inputInt("Enter the side length")
if s < 300:
    square(s)
else:
    print "The side length is too big"
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Les instructions du bloc *if* ne sont exécutées que si la condition est vraie et, dans le cas contraire, ce sont les instructions du bloc *else* qui sont exécutées. Le bloc *else* est facultatif.

Remarquez le double point obligatoire après la condition du *if* et après le *else*. D'autre part, les instructions prenant part aux blocs *if*` et *else* doivent être indentées de manière cohérente.

BRANCHEMENTS MULTIPLES

À présent, on aimerait dessiner des carrés colorés. L'utilisateur peut spécifier la couleur via un **nombre entier** saisi dans la boîte de dialogue.On décide à l'aide d'une **structure if** de la couleur de remplissage à utiliser en fonction du nombre saisi. On teste d'abord si le nombre entré est 1 avec le bloc *if*.

Si ce n'est pas le cas, on teste s'il vaut 2 dans le bloc *elif*, puis s'il vaut 3 de la même manière. Finalement, pour tout autre nombre que 1, 2 ou 3, on utilise le bloc *else* pour choisir la couleur noire par défaut.

L'instruction **fill(10, 10)** va remplir la surface entourant le point de coordonnées (10, 10) avec la couleur spécifiée avec *setFillColor()*. Du fait que la tortue se retrouve au centre de la fenêtre aux coordonnées (0, 0) après avoir dessiné le carré, il est certain que le point (10, 10) se trouve à l'intérieur du carré.


```
from gturtle import *

def square():
    repeat 4:
        forward(100)
        right(90)

makeTurtle()
n = inputInt("Enter a number: 1:red 2:green 3:yellow")
if n == 1:
    setFillColor("red")
elif n == 2:
    setFillColor("green")
elif n == 3:
    setFillColor("yellow")
else:
    setFillColor("black")

square()
fill(10, 10)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

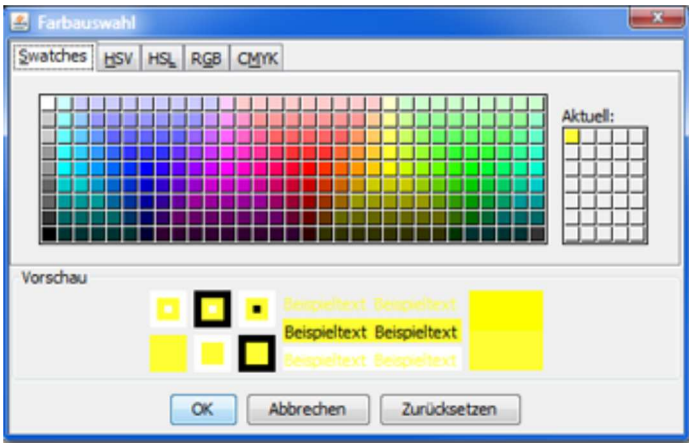
MEMENTO

Plusieurs conditions peuvent être évaluées de manière consécutive. Dans le cas où la condition du bloc *if* n'est pas vérifiée, l'interpréteur teste la condition du prochain bloc *elif* qui est une abréviation de *else if*. Si aucune des conditions *elif* n'est satisfaite, ce sont les instructions du bloc *else* qui sont exécutées. Une erreur de débutant courante dans l'écriture des conditions consiste à utiliser le signe = pour tester une égalité entre deux expressions au lieu du double égal ==. En effet, il faut se rappeler que le symbole = représente l'opérateur d'affectation à une variable et non l'opérateur de comparaison. Voici la notation utilisée pour les opérateurs de comparaison : >, >=, <, <=, ==, *!*=.

Finalement, la fonction *fill(x,y)* permet de remplir des figures géométriques fermées en utilisant la couleur *color* préalablement choisie à cet effet en appelant *setFillColor(color)*. Il faut cependant que le point (x,y) se trouve à l'intérieur de la figure.

CHOIX DE COULEUR, VARIABLES ALÉATOIRES

Pour remplir une figure *a posteriori* avec la fonction *fill()*, il faut que tous les pixels de son intérieur soient vierges de couleur. Pour créer de nouvelles figures pleines au-dessus de figures existantes, il faut utiliser conjointement les fonctions **startPath()**/**fillPath()**. Dans le programme suivant, on appelle la fonction **askColor()** faisant apparaître une boîte de dialogue sympathique permettant de choisir la couleur de l'étoile.



Pour dessiner les différentes étoiles, on utilise la fonction **star()**, qui prend comme paramètres la taille de l'étoile ainsi qu'un paramètre *fill* qui peut être vrai (*True*) ou faux (*False*) et qui détermine si l'étoile doit être remplie de couleur ou non.

```
from gturtle import *

makeTurtle()

def star(size, filled):
    if filled:
        startPath()
    repeat 9:
        forward(size)
        left(175)
        forward(size)
        left(225)
    if filled:
        fillPath()

clear("black")
repeat 5:
    color = askColor("Color selection", "yellow")
    if color == None:
        break
    setPenColor(color)
    setFillColor(color)
    setRandomPos(400, 400)
    back(100)
    star(100, True)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

La fonction **askColor()** prend deux paramètres permettant de spécifier le texte affiche dans la barre de titre ainsi que la couleur initialement sélectionnée. Lorsque l'utilisateur clique sur le bouton OK, la fonction *askColor* retourne la couleur sélectionnée tandis qu'un clic sur le bouton *Annuler* force la fonction à retourner la valeur spéciale *None*. Il est donc possible d'utiliser une structure *if* pour tester si l'utilisateur a annulé le choix de la couleur et réagir en conséquence en sortant de la boucle *repeat* par le mot-clé *break*.

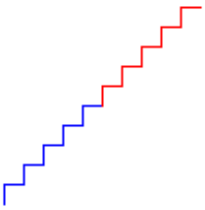
Une variable ou un paramètre qui prend les valeurs *True* ou *False* est qualifié de ***booléen*** [**plus...**]. Il n'est pas très élégant d'écrire *if filled == True :* puisque *filled* est une valeur booléenne. On peut simplement écrire *if filled if filled:*

EXERCICES

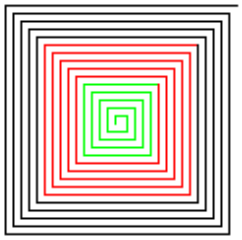
1. Dans une boîte de dialogue, demander à l'utilisateur de saisir la longueur du côté du carré à dessiner. Si elle est inférieure à 50, dessiner un carré rouge de la taille correspondante. Sinon, dessiner un carré vert de la taille indiquée.

2. Ordonner à la tortue de dessiner un escalier comprenant 10 marches en utilisant une instruction *repeat 10:*. Dessiner les 5 premières marches en bleu et les marches restantes en rouge (Figure a).

(a)



(b)



Demander à la tortue de dessiner une spirale en utilisant d'abord du vert, ensuite du rouge et, finalement, du noir (Figure b).

2.8 BOUCLES WHILE

INTRODUCTION

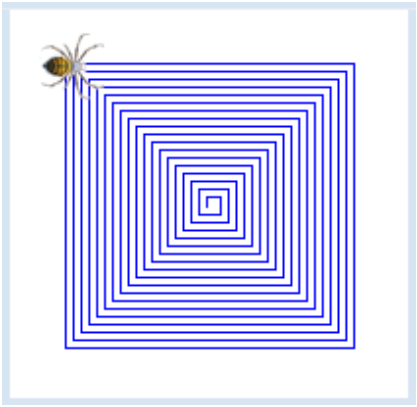
Nous avons déjà eu un premier contact avec les boucles au travers de la structure *repeat* qui permet de répéter un bloc d'instructions un nombre déterminé de fois. Il faut cependant savoir que cette structure *repeat* ne fait pas partie du langage Python standard mais constitue un ajout à l'environnement TigerJython dans un but purement pédagogique. Par contre, la boucle *while* est à disposition sur toutes les variantes de Python. Une boucle *while* débute par le mot-clé *while* suivi d'une condition qui va contrôler la répétition. Les instructions contenues dans le bloc *while* sont répétées aussi longtemps que la condition reste vérifiée. Une fois cette condition devenue fausse, la boucle s'interrompt et le programme continue normalement avec l'instruction qui suit directement le bloc d'instructions de la boucle.

CONCEPTS DE PROGRAMMATION: *Itérations, structure while, combinaisons de conditions, conditions d'arrêt d'une boucle.*

TOILE D'ARAIGNÉE

On aimerait que la tortue dessiner une spirale rectangulaire à l'aide d'une boucle *while*. On utilisera une variable *a* **initialisée à la valeur 5** qui sera **incrémentée de 2** à chaque ***itération*** (passage) de la boucle. Aussi longtemps que la **condition a < 200** est vraie, les instructions dans le bloc de la boucle seront exécutées.

Pour que ce soit un peu plus marrant, on peut remplacer la tortue par une araignée en spécifiant une autre image lors de l'appel de la fonction *makeTurtle()*.

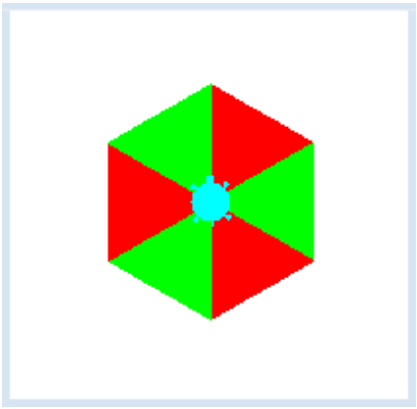


```
from gturtle import *

makeTurtle("sprites/spider.png")

a = 5
while a < 200:
    forward(a)
    right(90)
    a = a + 2
```

MEMENTO

Une boucle *while* est utilisée pour répéter un bloc de programme tant que la condition reste vraie. De ce fait, on pourrait la qualifier comme "*condition d'execution*". Si on néglige de changer la valeur de la variable contrôlant la condition (ici la variable *a*), la condition reste toujours vraie et le programme entre dans une boucle infinie sans jamais pouvoir en sortir. Comme cette erreur arrive à tout le monde au début, il faut savoir que l'on peut arrêter un programme bloqué dans une boucle infinie avec le bouton **stop** ou en fermant la fenêtre de la tortue. En général, les boucles infinies sont dangereuses si l'on n'a aucune d'interrompre le programme. Dans les cas les plus extrêmes, il faut redémarrer l'ordinateur.

COMBINER DES CONDITIONS AVEC *OR*



On aimerait dessiner la figure ci-contre avec la tortue à l'aide d'une **boucle while**. Comme le montre la figure, on aimerait alterner le rouge et le vert. Pour ce faire, on peut choisir la couleur sur la base de la valeur de la variable de contrôle, en coloriant en rouge lorsque la variable *i* prend les valeurs 0, 2 et 4.

La fonction **fillToPoint(0. 0)** permet de remplir la figure en la dessinant. Cela se comporte comme si un ruban était attaché au point (0,0) à une extrémité et à la tortue à l'autre extrémité et que tous les points touchés par le ruban étaient coloriés.

```
from gturtle import *

def triangle():
    repeat 3:
        forward(100)
        right(120)

makeTurtle()
i = 0
while i < 6:
    if i == 0 or i == 2 or i == 4:
         setPenColor("red")
    else:
         setPenColor("green")

    fillToPoint(0, 0)
    triangle()
    right(60)
    i = i + 1
```

MEMENTO

Il faut faire très attention à ce que les indentations des blocs soient correctement ajustées lorsque l'on utilise plusieurs structures de contrôle imbriquées (ici, le *if* à l'intérieur du *while*).

Comme on le voit dans le précédent programme, il est possible de combiner plusieurs conditions à l'aide de l'opérateur logique OU (*or*). Une telle condition est vraie si au moins une des conditions combinées est vraie.

La fonction *fillToPoint(x, y)* permet de colorier l'intérieur des formes dessinées par opposition à la fonction *fill()* permettant de colorier des figures fermées déjà existantes.

COMBINER DES CONDITIONS AVEC *AND*

On aimerait que la tortue dessine 10 maisons juxtaposées à l'aide d'une boucle *while*. Les maisons sont numérotées de 1 à 10. Les maisons 4 à 7 sont grandes et les autres petites. Dans la boucle *while*, la variable *nr* qui contrôle la boucle est utilisée pour déterminer la taille des maisons. Les maisons sont grandes si *nr* est supérieur à 3 **et** inférieur à 8.

On utilise la fonction **fillToHorizontal(0)** pour ajouter de la couleur à la surface comprise entre la ligne dessinée par la tortue (les toits) et la droite horizontale d'équation cartésienne y = 0.

```
from gturtle import *

makeTurtle()
setPos(-200, 30)
right(30)
fillToHorizontal(0)
setPenColor("sienna")

nr = 1
while nr <= 10:
    if nr > 3 and nr < 8:
        forward(60)
        right(120)
        forward(60)
        left(120)
    else:
        forward(30)
        right(120)
        forward(30)
        left(120)

    nr += 1
```

MEMENTO

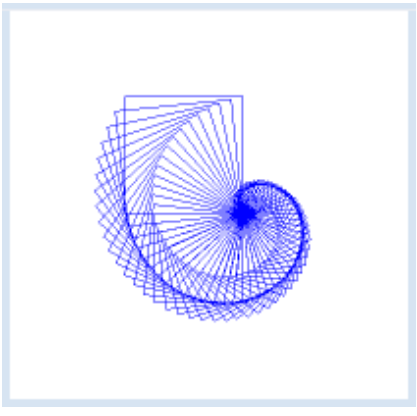
On peut combiner plusieurs conditions à l'aide de l'opérateur logique **and**. Une telle combinaison de conditions est alors vraie si et seulement si toutes les conditions qui la composent sont vraies.La fonction **fillToHorizontal(y)** permet de colorier la surface délimitée par la ligne dessinée par la tortue et la droite horizontale passant par les points d'ordonnée y.

L'expression **nr += 1** est un raccourci pour *nr = nr + 1*, ce qui a pour effet d'augmenter la variable *nr* de 1. On dit que l'on ***incrémente*** la variable *nr* de *1*.

INTERRUPTION DE BOUCLES AVEC *BREAK*

Une boucle dont la condition est toujours vraie va tourner à l'infini. Il est cependant possible d'interrompre l'exécution d'une boucle avec le mot-clé **break**.

Le programme suivant dessine des carrés de côtés croissants et tournés de 6° jusqu'à ce que la longueur du côté soit 120.



```
from gturtle import *

def square(sidelength):
    repeat 4:
        forward(sidelength)
        left(90)

makeTurtle()
```

```
hideTurtle()

i = 0
while 1 == 1:
    if i > 120:
        break
    square(i)
    right(6)
    i += 2
print "i =", i
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Au lieu d'utiliser une condition bidon toujours vraie telle que **while 1 == 1**. on peut écrire tout simplement *while True:* car *True* est une valeur booléenne toujours vraie et *False* est une valeur booléenne toujours fausse.

La boucle procède en incrémentant la variable de contrôle *i* par pas de *2*. On utilise le raccourci **i += 2** au lieu d'écrire *i = i + 2*.

L'instruction python **print** permet d'écrire dans la console de TigerJython au bas de l'éditeur. Les chaines de caractères doivent être spécifiées entre guillemets doubles " et les nombres doivent être séparés par des virgules. Un espace sera automatiquement rajouté par *print* entre le texte et le nombre.Êtes-vous capables d'expliquer pourquoi le programme affiche *i = 122* lorsqu'il sort de la boucle ?

Le mot-clé *continue* est rarement utilisé et permet d'interrompre l'exécution courante du bloc tout en reprenant une nouvelle itération après avoir vérifié la condition de la boucle.

VALIDATION DE SAISIE

Si l'on demande à l'utilisateur de saisir un nombre compris dans une certaine plage de valeurs, on ne doit pas avoir une confiance aveugle dans ses capacités ou sa volonté de bien se plier à nos exigences. Un programme robuste vérifie la validité des données saisies par l'utilisateur et traite les saisies incorrectes par un message adapté. Ce genre de validation est la plupart du temps effectué dans une boucle *while* interrompue uniquement lorsque la saisie utilisateur est acceptée. Dans le programme suivant, l'utilisateur doit entrer une des valeurs entières 1, 2 ou 3 afin de choisir la couleur rouge, verte ou jaune du disque plein.

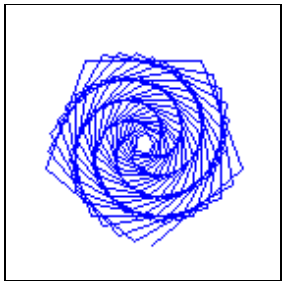
```
from gturtle import *

makeTurtle()

n = 0
while n < 1 or n > 3:
    n = inputInt("Enter 1, 2 or 3")
if n == 1:
    setPenColor("red")
elif n == 2:
    setPenColor("green")
else:
    setPenColor("yellow")
dot(200)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

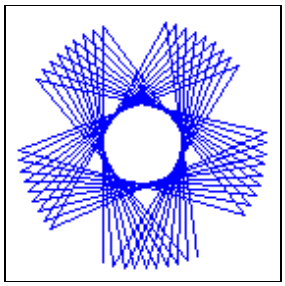
EXERCICES

1. La tortue avance sur un segment droit de longueur 5 puis tourne de 70° vers la droite. Ensuite, la longueur du segment est incrémentée de 0.5. Répéter ces étapes tant que la longueur du segment est inférieure à 150.

   Tester ensuite le programme avec une rotation de 89° au lieu de 70°.

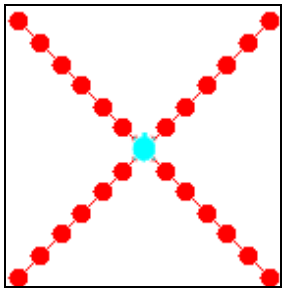


2. Comme vous l'avez sans doute remarqué, l'angle de rotation au sommet d'une branche était de 144° pour l'étoile à 5 branches. Appliquez un petit changement à cet angle et augmentez le nombre de répétitions de la boucle. Vous obtiendrez alors une nouvelle figure



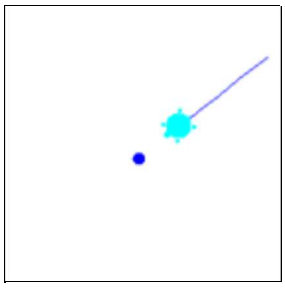
3. La tortue dessine un motif en diagonale avec des disques remplis de rouge. Tous les disques sont situés à l'intérieur de la fenêtre, ce qui implique que la distance de chaque point au centre de la fenêtre est inférieure à 400.

   Utiliser l'instruction *dot(25)* pour créer les disques.



4. La tortue est initialement positionnée au point (250, 200). Elle avance en ligne droite par pas de 10 en direction du centre de la fenêtre (0,0) jusqu'à ce que la distance qui la sépare du centre soit inférieure à 1.

   Utiliser les fonctions *towards() et heading(degrees)* (voir la **documentation)**.



5*. On veut modifier le programme de l'exercice précédent pour arrêter la tortue de manière plus précise sur le centre de l'écran (0,0). On réduit donc la condition d'arrêt (distance restante jusqu'au centre) d'un facteur 10 à 100. Il se peut étonnamment que pour certaines valeurs, la tortue ne s'arrête plus. Expliquer ce phénomène

2.9 RÉCURSIONS

INTRODUCTION

Vous vous rappelez peut-être de l'étrange histoire de l'homme à la dent creuse que l'on raconte volontiers aux petits enfants :

Äs isch ämal ä Ma gsi
dä het ä hohle Zahn gha
u i däm hohle Zahn isch äs Schachteli gsi
u i däm Schachteli isch äs Briefli gsi
u i däm Briefli isch gstande:
Äs isch ämal ä Ma gsi...

Il était une fois un monsieur
qui avait une dent creuse
Dans cette dent creuse, était une boîte
dans la boîte, était une lettre
sur cette lettre, il était écrit:
Il était une fois un monsieur... .

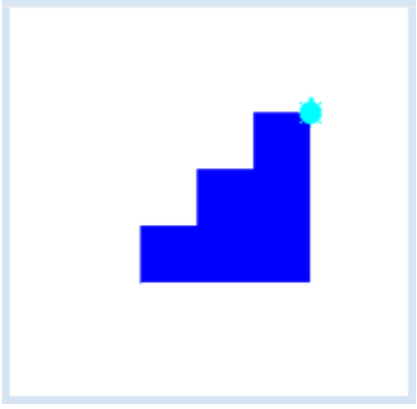
Des structures qui se font référence à elles-mêmes dans leur définition sont appelées ***récursives***. Les poupées russes ou matryoshkas en sont un exemple très connu : une matryoshka contient une autre matryoshka légèrement différente et plus petite qui contient elle-même une autre matryoshka qui contient elle-même …

CONCEPTS DE PROGRAMMATION: *Récursion, ancrage de récursion, récursion indirecte*

ESCALIERS RÉCURSIFS

Admettons qu'il nous soit demandé de construire un escalier de trois marches. Au lieu de construire trois marches l'une après l'autre, on peut imaginer un escalier de trois marches comme une marche de longueur 3 surmontée d'un escalier de deux marches. Cet escalier de deux marches est alors constitué d'une marche de longueur 2 surmontée d'un escalier à une marche.

Cet escalier à une marche peut être lui-même considéré comme une marche de longueur 1 surmontée d'un escalier à zéro marche. Un escalier à zéro marche n'est composé de rien du tout et l'on peut s'arrêter de construire.



Cette procédure de construction est récursive puisque la définition d'un escalier de 3 marches (*n* marches de manière plus générale) repose sur un escalier de 2 marches (*n-1* marches de manière plus générale). Sous forme de programme, cela donne:

```
def stairs(n):
    step()
    stairs(n-1)
```

Il ne reste plus qu'à instruire la tortue sur la façon de construire une marche de taille *n* en définissant la fonction *step()*. Il suffit ensuite d'appeler *stairs(3)* pour dessiner un escalier de trois marches. Notre programme ne fonctionnerait cependant pas en l'état, car il ne traite pas le cas

de base d'un escalier à zéro marche. On peut facilement régler ce problème avec une condition *if* :

```
if n == 0:
   return
```

L'instruction *return* interrompt l'exécution de la fonction, ce qui a pour effet d'interrompre le travail actuel et de retourner au travail en cours avant le dernier appel à *step()*. Voici à quoi devrait ressembler le programme après cet ajout indispensable:

```
from gturtle import *

def stairs(n):
    if  n == 0:
        return
    step()
    stairs(n - 1)

def step():
    forward(50)
    right(90)
    forward(50)
    left(90)

makeTurtle()
fillToHorizontal(0)
stairs(3)
```

MEMENTO

La récursion est une méthode de résolution de problème essentielle en mathématiques et en informatique consistant à résoudre un problème en le transformant un en problème de même nature mais légèrement simplifié. Ainsi, dans une récursion directe, si *f* est une fonction qui qui fournit la solution cherchée, *f* est réutilisée dans sa propre définition [**plus...**].

De prime abord, il peut paraître étrange de vouloir résoudre un problème par une méthode qui présuppose la connaissance de sa solution. Il faut noter cependant que la définition récursive ne présuppose pas la solution du même problème, mais d'un problème similaire de taille réduite, plus proche de la solution cherchée. Pour réduire la taille du problème, on définit la fonction récursive *f* avec un paramètre *n*

```
def f(n):
    ...
```

Bei der Wiederverwendung von f im Definitionsteil wird der Parameter verkleinert:

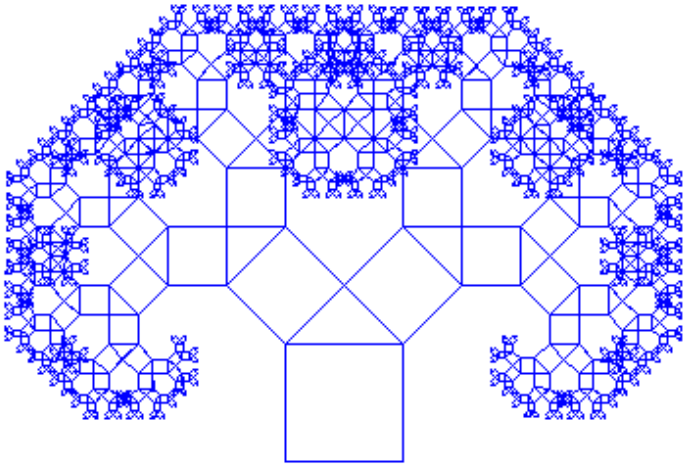
```
def f(n):
    ...
    f(n-1)
    ...
```

Une fonction ainsi définie s'appellerait récursivement et n'aboutirait jamais à la solution cherchée. Pour éviter cela, il est nécessaire de définir un critère de terminaison appelé ***ancrage de la récursion***.

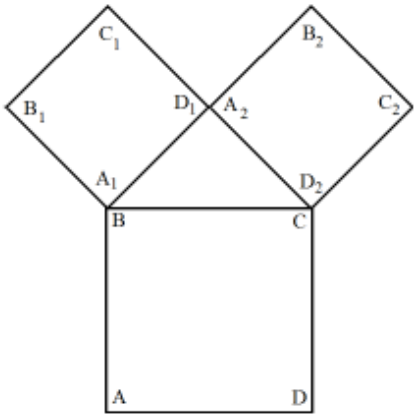
```
def f(n):
    if n == 0:
       return
    ...
    f(n - 1)
```

L'instruction *return* prévient tout traitement ultérieur de la fonction.

L'ARBRE DE PYTHAGORE



Les algorithmes récursifs permettent de dessiner de magnifiques dessins. Voici comment obtenir l'arbre ci-dessus:



- En partant du point A, dessiner un carré ABCD en s'assurant que sa base soit le côté AD
- Ajouter un triangle rectangle isocèle BD1C au-dessus du côté BC du carré
- Recommencer la procédure en prenant les cathètes du triangle rectangle isocèle comme bases des prochains carrés à dessiner

Il est bien établi que l'implémentation d'un programme récursif est inhabituelle. De ce fait, voici une procédure servant de guide dans l'élaboration de ce programme:

- Définir une fonction *square(s)* qui ordonne à la tortue de dessiner un carré de côté *s* et qui s'assure qu'elle achève le dessin dans la même position et la même orientation qu'au début du dessin
- Définir une fonction *tree(s)* permettant de dessiner un arbre en commençant par un carré dont la longueur du côté est *s*. Cette fonction peut s'appeler elle-même de manière récursive. Il est capital qu'après avoir dessiné l'arbre en question, la tortue se retrouve dans le même position et avec la même orientation que lorsqu'elle a débuté le dessin de cet arbre. Il est toujours judicieux dans ces cas de se mettre à la place de la tortue pour bien comprendre le programme. À chaque étape, les lignes ajoutées sont mises en évidence

- On commence par dessiner le carré dont le côté mesure *s* en partant du point A

```
def tree(s):
    square(s)
```

- Ensuite, se rendre au point B, tourner de 45 degrés vers la gauche et considérer cette position et cette orientation comme le point de départ pour le dessin d'un prochain sous-arbre de longueur réduite *s1*. D'après le théorème de Pythagore, on a:

$$s1 = \frac{s}{\sqrt{2}}$$

```
def tree(s):
    square(s)
    forward(s)
    s1 = s / math.sqrt(2)
    left(45)
    tree(s1)
```

- Étant donné qu'après avoir dessiné un arbre, on se retrouve dans la position et l'orientation de départ, on se retrouve au point B avec une orientation de 45 degrés

```
def tree(s):
    square(s)
    forward(s)
    s1 = s / math.sqrt(2)
```

vers la gauche (Nord-Ouest) dans la direction du sommet B1

En se tournant de 90 degrés et en avançant d'une distance *s1*, la tortue se retrouve au point D1, orientée vers B2. Il s'agit de la condition initiale permettant de redessiner un arbre avec *tree(s1)*

```
left(45)
tree(s1)
right(90)
forward(s1)
tree(s1)
```

- Finalement, il est nécessaire de retourner au point initial A en adoptant l'orientation de départ. Pour ce faire, il suffit de reculer sur une distance *s1*, de tourner de 45 degrés vers la gauche et de reculer sur une distance *s*.

```
def tree(s):
    square(s)
    forward(s)
    s1 = s / math.sqrt(2)
    left(45)
    tree(s1)
    right(90)
    forward(s1)
    tree(s1)
    back(s1)
    left(45)
    back(s)
```

```
from gturtle import *
import math

def tree(s):
    if s < 2:
        return
    square(s)
    forward(s)
    s1 = s / math.sqrt(2)
    left(45)
    tree(s1)
    right(90)
    forward(s1)
    tree(s1)
    back(s1)
    left(45)
    back(s)

def square(s):
    repeat 4:
        forward(s)
        right(90)

makeTurtle()
ht()
setPos(-50, -200)
tree(100)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Lorsqu'on dessine des motifs récursifs avec la tortue, il est très important qu'elle retrouve sa position et son orientation initiale après avoir effectué le dessin.

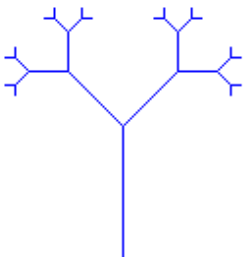
EXERCICES

1. Expliquer la différence essentielle entre ces deux programmes. Examiner en particulier la position et l'orientation de la tortue à la fin du programme. Pourquoi *figA* est-elle appelée « Last Line Recursion » alors que *figB* est-elle appelée « First Line Recursion »?

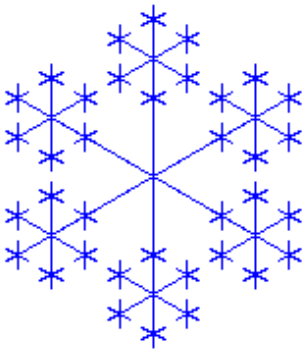
```
from gturtle import *

def  figA(s):
   if s > 200:
      return
   forward(s)
   right(90)
   figA(s + 10)
makeTurtle()
figA(100)
```

```
from gturtle import *

def  figB(s):
   if s > 200:
      return
   figB(s + 10)
   forward(s)
   right(90)
makeTurtle()
figB(100)
```

2. L'arbre binaire complet est un graphe bien connu. La figure ci-contre représente un arbre binaire de profondeur 4. Le tronc unique est rajouté en plus et n'en fait pas partie à strictement parler. Définir une fonction récursive *tree(s)* qui dessine un arbre binaire complet (sans le tronc qu'il faut rajouter *a posteriori*) de taille *s*.

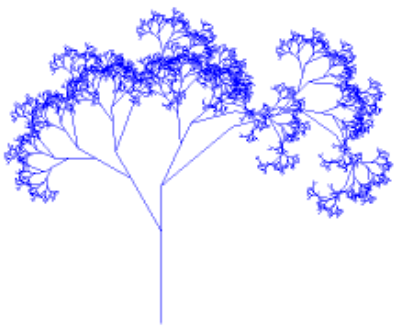


3. Dessiner l'étoile ci-contre en définissant la fonction récursive *star(s)* où *s* représente la taille de l'étoile, à savoir la distance entre le centre de l'étoile et le centre des étoiles de la génération suivante. *s* diminue à 1/3 de sa valeur précédente. Dessiner l'étoile avec l'appel *star(180)* et définir l'ancrage de la récursion de telle sorte qu'elle s'arrête lorsque *s* < *20*. En cachant la tortue avec *hideTurtle()*, le dessin se fera bien plus rapidement.



4*. Maintenant, dessinez un arbre qui ressemble presque à un arbre réaliste. Dans ce but, définir une fonction *treeFractal(s)* dont la tige mesure *s* construit de la manière suivante:

- Définir l'ancrage de la récursion de telle sorte qu'elle s'arrête lorsque la longueur *s* de la tige est inférieur à 5
- Avant de commencer le dessin de l'arbre, sauver les coordonnées *x* et *y* avec les fonctions *getX()* et *getY()* ainsi que son orientation avec *heading()* de manière à pouvoir les restaurer facilement à la fin du dessin
- Avancer de *s / 3*, tourner de 30 degrés vers la gauche et dessiner un arbre de taille *2*s/3*
- Tourner de 30 degrés vers la droite, avancer de *s / 6*, tourner de 25 degrés vers la droite et dessiner l'arbre de taille *s/2*
- Tourner encore une fois de 25 degrés vers la droite, avancer de *s / 3* et dessiner à nouveau un arbre de taille *n / 2*
- Restaurer les coordonnées et l'orientation initiales avec les fonctions *setPos(x, y)* et *heading(angle)*.



2.10 CONTRÔLE PAR LES ÉVÉNEMENTS

INTRODUCTION

Jusqu'à présent, on a abordé uniquement des programmes ne comportant qu'un seul fil d'événements, les instructions étant exécutées les unes après les autres dans l'ordre du programme avec des boucles et des conditions. Ainsi, le fil d'exécution du programme est complètement déterminé par les paramètres initiaux. Ce modèle ne convient pas bien au pilotage d'un programme avec une souris car on ne peut pas savoir où en est l'exécution du programme lors d'un clic de souris. Pour pouvoir capturer les clics de souris, il faut introduire un nouveau concept de programmation appelé ***contrôle par les événements*** ou ***programmation événementielle***. Le principe en est le suivant :

Il faut définir une fonction portant un nom quelconque, par exemple *onMouseHit()* qui n'est jamais appelée explicitement par le programme. Ensuite, on demander à l'ordinateur d'appeler cette fonction à chaque fois qu'il détecte un clic de souris. En résumé, on demande au programme d'exécuter la fonction *onMouseHit()* à chaque fois que l'utilisateur clique sur le bouton de la souris.

CONCEPTS DE PROGRAMMATION : *Programmation événementielle, événements de la souris.*

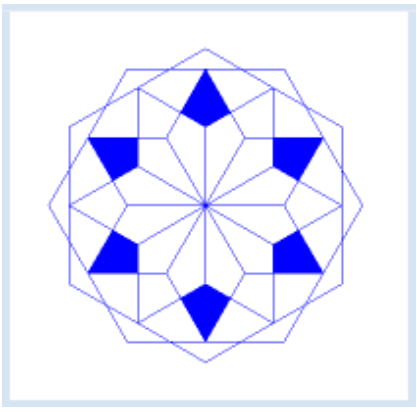
ÉVÉNEMENTS DE LA SOURIS

Il est très facile d'implémenter ce nouveau concept en Python. Dans notre premier programme dirigé par les événements, la tortue doit dans un premier temps dessiner une figure sympathique. Ensuite, l'utilisateur peut décorer ce dessin initial en cliquant avec la souris sur certaines zones pour les colorier.

Le programme ci-dessous définit la fonction **onMouseHit(x, y)**, qui sera appelée par le système en lui fournissant en guise de paramètres les coordonnées de la souris lors de clic. On peut ensuite utiliser ces coordonnées pour colorier la figure fermée qui entoure le point *(x, y)* à l'aide de la fonction **fill(x, y)**.

Il est également capital d'enregistrer notre gestionnaire d'événements de souris *onMouseHit()* auprès du système. Cela est possible en renseignant le paramètre **mouseHit** de la fonction *makeTurtle()* avec une référence à notre gestionnaire d'événemnts *onMouseHit*.

Le programme utilise *hideTurtle()* pour accélérer le dessin.



```
from gturtle import *

def onMouseHit(x, y):
    fill(x, y)

makeTurtle(mouseHit = onMouseHit)
hideTurtle()
```

```
addStatusBar(30)
setStatusText("Click to fill a region!")

repeat 12:
    repeat 6:
        forward(80)
        right(60)
    left(30)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Du point de vue technique, la programmation événementielle est implémentée en définissant une fonction appelée ***gestionnaire d'événements*** (*event handler* en anglais) qui sera appelée sans que l'on s'en occupe à chaque fois que le système d'exploitation détecte un certain type d'événement. Pour que cela fonctionne, il faut informer le système de notre gestionnaire d'événement en passant le nom de la fonction (sans parenthèse !!!) au paramètre approprié de la fonction *makeTurtle()*. Pour cela, on utilise la notion très utile en Python de paramètre nommé avec la notation *parameter_name = parameter_value*.

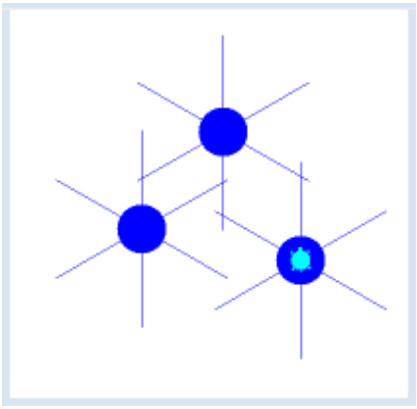
On peut régler les préférences pour la couleur de remplissage avec la fonction *setFillColor()*.

La fonction **addStatusBar(n)** permet d'afficher des informations utiles pour l'utilisateur dans une barre d'état en-dessous de la fenêtre de la tortue. Le nombre entier *b* permet de spécifier la hauteur de ligne du texte affiché dans la barre d'état.

DESSINER AVEC UN ÉVÉNEMENT DE LA SOURIS

Dessinons une étoile possédant des rayons à partir de la position du clic de souris. Pour ce faire, on définit la fonction **onMouseHit(x, y)**, où l'on explique la tortue comme dessiner l'étoile.

Pour que la fonction *onMouseHit()* soit invoquée par le système lors des clics de souris, on passe au paramètre **mouseHit** de la fonction *makeTurtle()* une référence vers notre fonction *onMouseHit* (remarquer l'absence de parenthèses).



```
from gturtle import *

def onMouseHit(x, y):
    setPos(x, y)
    repeat 6:
        dot(40)
        forward(60)
        back(60)
        right(60)

makeTurtle(mouseHit = onMouseHit)
speed(-1)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

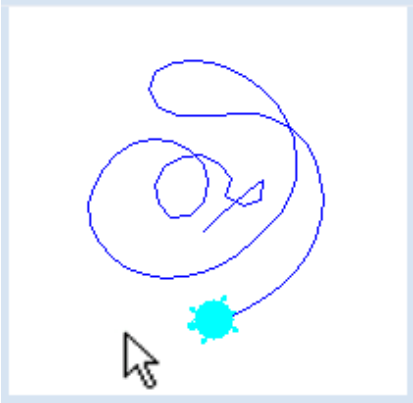
Ce programme est bogué : si l'utilisateur clique une seconde fois avant que l'étoile précédente ne soit terminée, l'étoile en cours ne va pas être terminée et le programme commence directement le dessin de la seconde étoile. De plus, les instructions non exécutées de l'étoile précédente vont être exécutées dans la deuxième étoile, ce qui mènera à un dessin incorrect de la deuxième étoile.

Ce comportement erroné est apparemment dû au fait qu'à chaque clic de la souris, la fonction *onMouseHit()* est appelée et exécutée, même si l'exécution précédente n'est pas encore terminée. Pour prévenir ce problème, il faut passer la fonction *onMouseHit* au paramètre *mouseHitX* au lieu de *mouseHit* lors de l'appel de *makeTurtle()*.

CHASSE À LA SOURIS

On aimerait que la tortue suive la souris partout où elle va. On ne peut pas utiliser l'événement du clic de la souris pour cela, car c'est le mouvement de la souris qui nous intéresse. La fonction *makeTurtle()* reconnaît le paramètre *mouseMoved* auquel on peut passer la référence vers une fonction qui sera appelée à chaque déplacement de la souris avec les nouvelles coordonnées.

La fonction **onMouseMoved(x, y)** reçoit les nouvelles coordonnées x et y de la souris.



```
from gturtle import *

def onMouseMoved(x, y):
    setHeading(towards(x, y))
    forward(10)

makeTurtle(mouseMoved = onMouseMoved)
speed(-1)
```

MEMENTO

En plus de *mouseHit* et *mouseHitX*, la fonction *makeTurtle()* met d'autres paramètres à disposition permettant de détecter des événements de la souris. Ces événements, au lieu de prendre les coordonnées *x* et *y* comme paramètres, prennent un seul paramètre *event* à l'aide duquel on peut déterminer les coordonnées de la souris lors de l'événement ainsi que d'autres informations.

mousePressed	Le bouton de la souris est enfoncé
mouseReleased	Le bouton de la souris est relâché
mouseClicked	Le bouton de la souris est enfoncé et relâché directement (clic)
mouseDragged	La souris se déplace pendant que le bouton est enfoncé
mouseMoved	La souris a été déplacée
mouseEntered	La souris entre dans la fenêtre de la tortue
mouseExited	La souris sort de la fenêtre de la tortue

Il est également possible d'enregistrer plusieurs gestionnaires d'événements différents en spécifiant simultanément plusieurs paramètres, par exemple les deux fonctions *onMousePressed()* et *onMouseDragged* :

```
makeTurtle(mousePressed = onMousePressed, mouseDragged = onMouseDragged)
```

On peut déterminer lequel des boutons a été cliqué grâce aux fonctions *isLeftMouseButton()* et *isRightMouseButton()*.

Les événements listés ci-dessus comportent une différence importance avec *mouseHit* : le mouvement de la tortue n'est pas visible pendant l'exécution de la fonction. De ce fait, il faut soit régler la vitesse de la tortue sur ultrarapide avec *speed(-1)* et cacher la tortue avec *hideTurtle()* ou alors exécuter le code du mouvement dans la partie principale du programme.

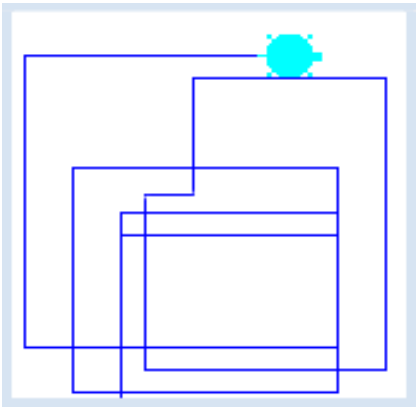
```
from gturtle import *

def onMousePressed(x, y):
    moveTo(x, y)

makeTurtle(mousePressed = onMousePressed)
#speed(-1)
#hideTurtle()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

ÉVÉNEMENTS DU CLAVIER

À chaque fois qu'une touche du clavier est actionnée, un événement est généré. Pour gérer ce dernier, on enregistre un gestionnaire d'événements en renseignant le paramètre **keyPressed**. lors de l'appel de la fonction *makeTurtle*.

Le gestionnaire d'événements reçoit alors en guise de paramètre un nombre entier qui identifie la touche actionnée. Il est possible de découvrir le code généré par chacune des touches du clavier en affichant le nombre entier dans la console avec *print* et en réalisant quelques expérimentations. Dans le programme suivant, la tortue se déplace de 10 pas en avant tant que le programme tourne. En actionnant les touches directionnelles du clavier, l'utilisateur peut changer son orientation selon les quatre orientations cardinales. Pour éviter que la souris ne quitte la fenêtre, le mode wrap est activé avec l'appel *wrap()*.



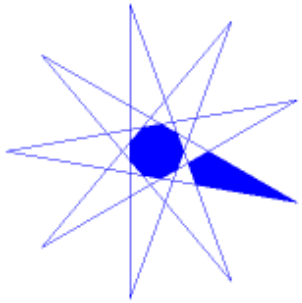
```
from gturtle import *

LEFT = 37
RIGHT = 39
UP = 38
DOWN = 40

def onKeyPressed(key):
    if key == LEFT:
        setHeading(-90)
    elif key == RIGHT:
        setHeading(90)
    elif key == UP:
        setHeading(0)
    elif key == DOWN:
        setHeading(180)
```

```
makeTurtle(keyPressed = onKeyPressed)
wrap()
while True:
    forward(10)
```

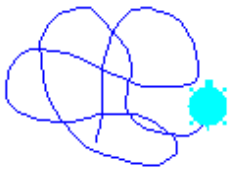
Ctrl+C pour copier, Ctrl+V pour coller (Ctrl+C pour copier, Ctrl+V pour coller)

EXERCICES

1. Dessiner l'étoile ci-contre à l'aide d'une boucle et la remplir ensuite avec des clics de souris pour la personnaliser.



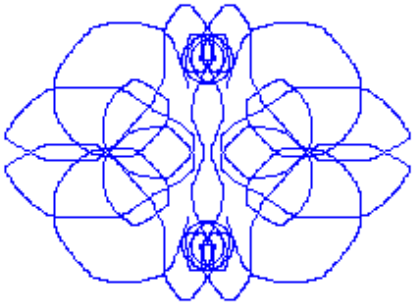
2. Réaliser un petit programme de dessin à l'aide de la tortue permettant de dessiner librement avec la souris. Pour cela, abaisser le crayon de la tortue lorsque le bouton de la souris est enfoncé, utiliser l'événement *mouseDrag* pour tracer les mouvements de la souris et relever le crayon lorsque le bouton est relâché.



3. Compléter le programme précédent en utilisant le bouton gauche de la souris pour dessiner une figure libre et le bouton droit pour colorier des surfaces fermées formées par la trace dessinée.

4. Utiliser l'exercice 2 pour créer un programme pour dessiner une figure kaléidoscopique en déplaçant la souris avec la touche pressée. La tortue (non visible) trace une ligne simultanément dans les quatre quadrants. Pour dessiner un segment de line, utiliser la fonction suivante:

```
def line(x1, y1, x2, y2):
    setPos(x1, y1)
    moveTo(x2, y2)
```



5. Modifier le programme avec les évènements du clavier de sorte qu'une région fermée dans laquelle se trouve la tortue est colorée en pressant la touche espace (code 32). Utiliser le programme pour dessiner une lettre colorée quelconque.

BONUS

CURSEUR PERSONNALISÉ

On peut sans problème changer la représentation du curseur de souris en spécifiant une image quelconque, ce qui permet de personnaliser le programme. Pour ce faire, il faut appeler la fonction *setCursor()* et lui passer une des valeurs de la table ci-dessous. Il est même possible de spécifier une image personnelle avec la fonction **setCustomCursor()**. Un curseur de souris est une image au format PNG ou GIF de 32x32 pixels et possédant un fond transparent. Les curseurs *pencil.gif* et *cutemouse.gif* sont déjà présents dans la distribution de TigerJython dans le dossier *sprites*.

Pourquoi ne pas décorer le programme montré précédemment avec l'image *cuteturtle* ou votre propre image. Il faut s'assurer que la tortue se déplace toujours vers la souris en utilisant **moveTo()**.

```
from gturtle import *

def onMouseMoved(x, y):
      moveTo(x, y)

makeTurtle(mouseMoved = onMouseMoved)
setCustomCursor("sprites/cutemouse.gif")
speed(-1)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

L'appel **speed(-1)** évite d'utiliser les animations de la tortue, ce qui rend les dessins avec *moveTo()* bien plus rapides.

Paramètres possible pour la fonction *setCursor()*:

Parameter	Icon
Cursor.DEFAULT_CURSOR	Icône par défaut
Cursor.CROSSHAIR_CURSOR	Icône en forme de viseur
Cursor.MOVE_CURSOR	Curseur de déplacement (Flèches en forme de croix)
Cursor.TEXT_CURSOR	Curseur de saisie de texte (ligne verticale)
Cursor.WAIT_CURSOR	Curseur d'attente

Le dossier *sprites* utilisé pour spécifier à la fonction *setCustomCursor()* le chemin vers le curseur doit se trouver dans le même dossier que le programme Python.

2.11 OBJETS TORTUES

INTRODUCTION

Dans la nature, une tortue est un individu ayant sa propre identité spécifique. Lors d'une exposition au zoo, on pourrait donner à chaque tortue son propre nom, par exemple, Pepe ou Maya. Cependant, les tortues ont également des caractéristiques communes: elles sont des reptiles appartenant à la classe animale des tortues. Cette notion de classes et d'individus a eu tant de succès qu'elle a été introduite en informatique comme une notion fondamentale, appelée programmation orientée objets (POO). Il vous sera facile d'apprendre les principes fondamentaux de la programmation orientée objet à l'aide des graphiques de tortue.

CONCEPTS DE PROGRAMMATION: *Classe, objet, programmation orientée objets, constructeur, clones.*

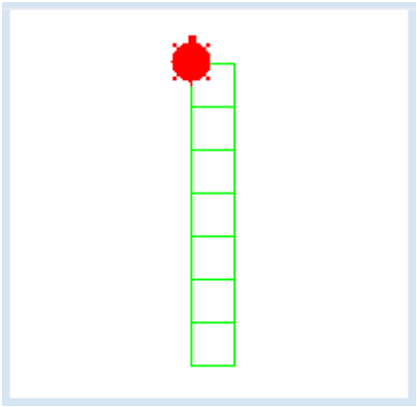
CRÉER UN OBJET TORTUE

La tortue était jusqu'à présent utilisée comme un objet anonyme auquel ne se référait aucune variable. Pour utiliser plusieurs tortues en même temps, il faut par contre donner à chaque tortue une identité propre en la nommant et en s'y référant à l'aide d'une variable.

L'instruction **maya = Turtle()** crée une nouvelle tortue nommée *maya*.
L'instruction *pepe = Turtle()*crée une nouvelle tortue nommée *pepe*.

Vous pouvez contrôler ces tortues nommées à l'aide des commandes que vous connaissez déjà, mais il faut toujours préciser explicitement la tortue à laquelle on s'adresse. Pour ce faire, on écrit le nom de la tortue cible tout d'abord, suivi d'un point et enfin la commande, par exemple **maya.forward(100)**.

Dans l'exemple suivant, maya dessine une échelle. Il n'est pas nécessaire d'écrire la ligne *makeTurtle()* plus puisque chaque tortue est crée ensuite à la main.



```
from gturtle import *

maya = Turtle()
maya.setColor("red")
maya.setPenColor("green")
maya.setPos(0, -200)

repeat 7:
    repeat 4:
        maya.forward(50)
        maya.right(90)
    maya.forward(50)
```

Sélectionner tout le code (Ctrl+C pour copier, Ctrl+V pour coller)

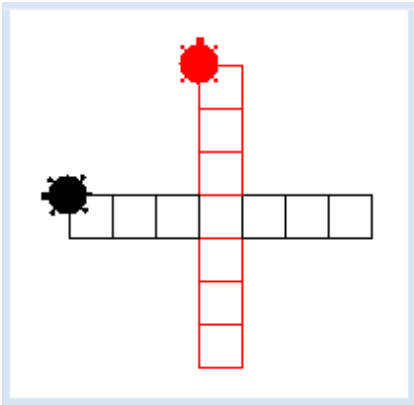
MEMENTO

Les objets similaires sont groupés par classes. Les objets d'une certaine classe sont fabriqués (on dit aussi instanciés) en utilisant le nom de la classe suivi de parenthèses. On parle alors du constructeur de la classe. Dans la suite, nous appellerons les fonctions rattachées à une certaine classe de méthodes.

CRÉER DAVANTAGE DE TORTUES

Nous savons maintenant qu'il est possible d'utiliser plusieurs tortues dans le même programme comme précédemment décrit. Pour créer *maya* et *pepe*, il faut simplement
maya = Turtle() et **pepe = Turtle()**

Ces deux tortues apparaissent cependant chacune dans une fenêtre séparée. Pour qu'elles apparaissent dans la même fenêtre, il faut d'abord leur créer un enclos commun en tant qu'objet de la classe *TurtleFrame*:
tf = TurtleFrame()
Cet objet doit ensuite être transmis au constructeur de la classe *Turtle* lors de la création des tortues. Dans le programme suivant, *maya* construit la même échelle que dans le programme précédent mais *pepe* construit une échelle noire horizontale.



```
from gturtle import *

tf = TurtleFrame()

maya = Turtle(tf)
maya.setColor("red")
maya.setPenColor("red")
maya.setPos(0, -200)

pepe = Turtle(tf)
pepe.setColor("black")
pepe.setPenColor("black")
pepe.setPos(200, 0)
pepe.left(90)

repeat 7:
    repeat 4:
        maya.forward(50)
        maya.right(90)
        pepe.forward(50)
        pepe.left(90)
    maya.forward(50)
    pepe.forward(50)
```

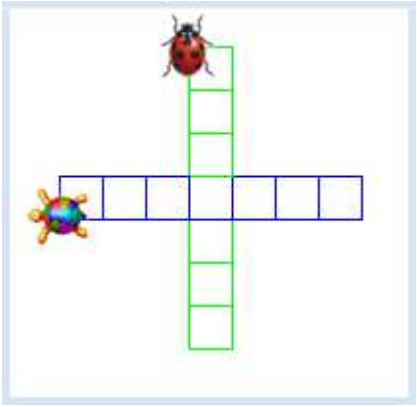
Sélectionner tout le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Pour placer plusieurs tortues dans la même fenêtre, il faut d'abord créer une instance de la classe *TurtleFrame* et passer cette dernière aux constructeurs de tortues. Lorsque deux tortues se trouvent au même point, il n'y a pas de collision car elles se superposent dans l'ordre de création. La dernière arrivée se trouve donc toujours au-dessus des précédentes.

DES PARAMÈTRES POUR LA TORTUE

Les objets tortues peuvent également être passés en paramètre lors des appels de fonctions. Puisque le même code est utilisé pour dessiner les échelles quelle quoi soit la tortue, il est indiqué de définir la fonction **step()** où *t* est le paramètre formel qui permet de spécifier la tortue qui devra obéir aux instructions dessinant une marche de l'échelle. Ensuite, on appelle deux fois cette fonction *step()*, une première fois en lui passant **maya** en tant que tortue à qui donner les ordres et une seconde fois en lui passant **pepe**.



```
from gturtle import *

def step(t):
    repeat 4:
        t.forward(50)
        t.right(90)
    t.forward(50)

tf = TurtleFrame()

maya = Turtle(tf, "sprites/beetle.gif")
maya.setPenColor("green")
maya.setPos(0, -150)
pepe = Turtle(tf, "sprites/cuteturtle.gif")
pepe.setPos(200, 0)
pepe.left(90)

repeat 7:
    step(maya)
    step(pepe)
```

Sélectionner tout le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

On peut personnaliser l'image utilisée pour chacune des tortues en spécifiant le chemin vers le fichier image lors de la création de la tortue. L'exemple précédent fait usage de deux fichiers images *beetle.gif* et *cuteturtle.gif* qui se trouvent dans la distribution de *TigerJython*.

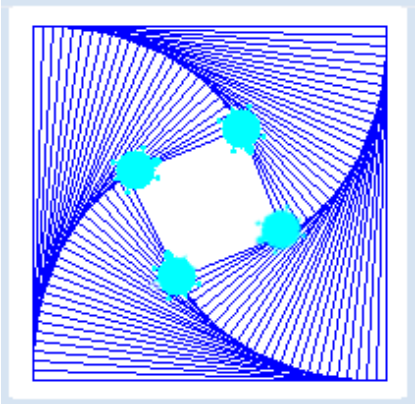
PROBLÈME DES SOURIS AVEC DES CLONES DE TORTUES

Dans le fameux problème des souris [**plus...**], *n* souris sont placées initialement aux sommets d'un polygone régulier de *n* côtés et se poursuivent les unes les autres à vitesse constante. La position des souris est fixée à intervalles de temps réguliers et chacune est alors tournée pour pointer la position de la souris qu'elle pourchasse. Une fois qu'elles ont toutes été orientées, elles avancent toutes d'une petite distance fixée.

On peut facilement résoudre ce problème par simulation en dessinant d'abord le polygone régulier à l'aide de la tortue globale puis en plaçant une nouvelle tortue à chaque sommet du polygone par clonage. Dans l'exemple suivant, on a posé *n* = *4*, ce qui correspond à un carré et placé les tortues nommées *t1*, *t2*, *t3* et *t4* au sommets du polygone avec la fonction **clone().** Un clone est un nouvel objet avec des propriétés identiques à l'objet cloné.

Ensuite, à l'intérieur d'une boucle, on ajuste leur position avec **setHeading()** et on les fait avancer d'un petit pas de cinq. La figure est particulièrement intéressante si l'on dessine les lignes « de visée » de chaque souris.

Le plus simple pour réaliser ces lignes est de définir une fonction **drawLine(a, b)**, qui fait aller la souris se trouvant en *a* jusqu'à la souris *b* et retour à l'aide de **moveTo()**.



```
from gturtle import *

s = 360

makeTurtle()
setPos(-s/2, -s/2)

def drawLine(a, b):
    ax = a.getX()
    ay = a.getY()
    ah = a.heading()
    a.moveTo(b.getX(), b.getY())
    a.setPos(ax, ay)
    a.heading(ah)

# generate Turtle clone
t1 = clone()
t1.speed(-1)
forward(s)
right(90)
t2 = clone()
t2.speed(-1)
forward(s)
right(90)
t3 = clone()
t3.speed(-1)
forward(s)
right(90)
t4 = clone()
t4.speed(-1)
forward(s)
right(90)
hideTurtle()

repeat:
    t1.setHeading(t1.towards(t2))
    t2.setHeading(t2.towards(t3))
    t3.setHeading(t3.towards(t4))
    t4.setHeading(t4.towards(t1))

    drawLine(t1, t2)
    drawLine(t2, t3)
    drawLine(t3, t4)
    drawLine(t4, t1)

    t1.forward(5)
    t2.forward(5)
    t3.forward(5)
    t4.forward(5)
```

Sélectionner tout le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

La fonction **clone()** permet de créer une nouvelle tortue à partir de la tortue globale de sorte qu'elle ait la même position, la même orientation et la même couleur. Dans le cas où la tortue originale est représentée à l'écran par une image personnalisée, le clone sera représenté par la même image. [**plus...**].

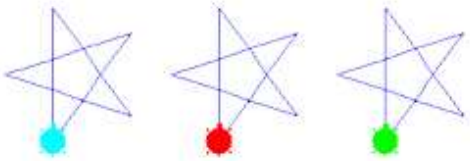
La fonction *drawLine()* peut être simplifiée en sauvegardant la position et l'orientation de la tortue avec *pushState()* et en les restaurant avec *popState()*:

```
def drawLine(a, b):
    a.pushState()
    a.moveTo(b.getX(), b.getY())
    a.popState()
```

On peut déterminer analytiquement les courbes des trajectoires engendrées (**Voir cet article**).

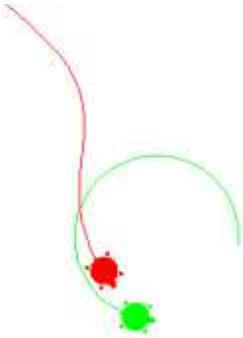
EXERCICES

1. Trois tortues doivent dessiner des étoiles à cinq branches simultanément. Chacune doit tracer un segment à tour de rôle. Elles doivent être de couleur cyan (la couleur par défaut), rouge et verte respectivement. La couleur sera spécifiée dans un paramètre additionnel du constructeur.



2. Une maman tortue verte tourne en rond à vitesse constante. Au début, la position du bébé tortue rouge est éloignée de la maman. Au fil du temps, le bébé tortue avance par petits pas en direction de sa maman.

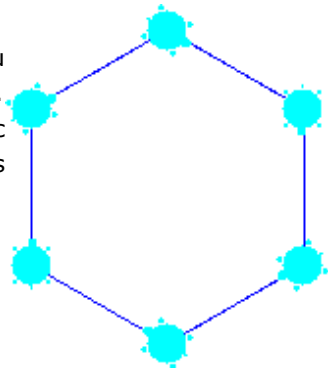
   Le bébé tortue, nommé *child* peut déterminer la direction pointant vers sa maman avec *direction = child.towards(mother)*

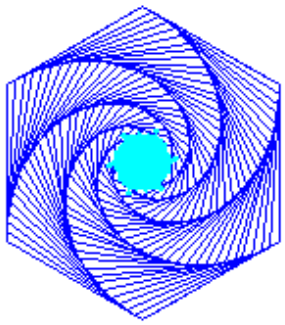


3. Laura dessine des carrés non vides. Après que chaque carré est terminé, une seconde tortue saute à l'intérieur et le remplit de vert.

   Utiliser une image différente pour chacune des tortues. Les images disponibles dans *tigerjython.jar* sont *beetle.gif*, *beetle1.gif*, *beetle2.gif* et *spider.png*. Vous pouvez également utiliser vos propres images qui doivent impérativement se trouver dans un dossier *sprites* situé dans le même dossier que votre script Python.

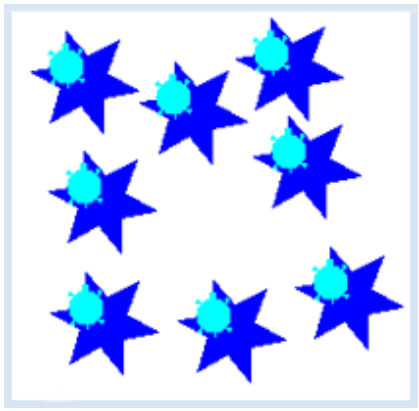
4. Représenter le graphique du « problème des souris » pour $n = 6$. Chacune des six souris débute donc sa trajectoire sur l'un des sommets de l'hexagone régulier.





MATÉRIEL BONUS

CRÉER DES TORTUES PAR DES CLICS DE SOURIS



Le programme ci-dessous dessine, lors de chaque clic de souris, une nouvelle tortue qui va tracer une étoile à partir du point situé aux coordonnées du clic, indépendamment des autres tortues. Ce programme permet de se rendre compte de la beauté et de la puissance de la programmation orienté objets de la programmation événementielle.

Pour gérer le clic de la souris, on définit la fonction **drawStar()**. Pour que cette fonction soit appelée par le système en réponse aux clics de la souris, il faut renseigner le paramètre **mouseHit**du constructeur de *TurtleFrame* avec le nom du gestionnaire d'événements *drawStar()*.

```
from gturtle import *

def drawStar(x, y):
    t = Turtle(tf)
    t.setPos(x, y)
    t.fillToPoint(x, y)
    for i in range(6):
        t.forward(40)
        t.right(140)
        t.forward(40)
        t.left(80)

tf = TurtleFrame(mouseHit = drawStar)
```

Ctrl+C pour copier, Ctrl+V pour coller (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Les objets qui ont les mêmes fonctionnalités et des propriétés similaires sont définis comme des classes dans le paradigme de la programmation orientée objets (OOP = Object Oriented Programming). Pour créer des objets particuliers, on utilise le constructeur (**Instanzen**).

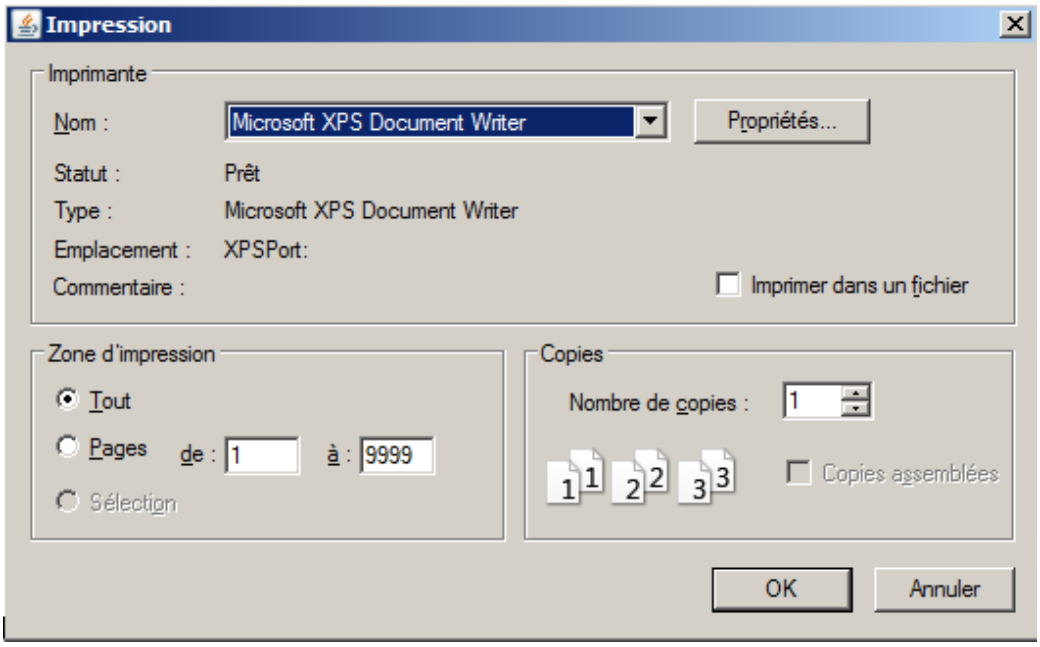
Pour traiter un clic de la souris, il faut écrire une fonction nommée de manière arbitraire mais pertinente qui comprend comme paramètres les coordonnées x et y du clic. Il faut ensuite enregistrer ce gestionnaire d'événements auprès du système en passant le nom de la fonction (gestionnaire d'événements) comme paramètre au constructeur de la classe *TurtleFrame*.

2.12 IMPRESSION

INTRODUCTION

Pour atteindre un meilleur rendu graphique on peut faire recours à l'impression puisque la résolution des imprimantes, typiquement 1200x1200 dpi, est bien meilleure que celle des écrans qui sont souvent limités à 100 dpi. L'impression de graphiques *GPanel* est réalisée de telle manière que les opérations de dessin sont imprimées sur papier au lieu d'être affichées à l'écran, ce qui engendre des graphiques de type vectoriels d'excellente qualité à l'impression. Pour imprimer un graphique, il faut commencer par définir une fonction nommée de façon arbitraire comportant les instructions du graphique souhaité. Lorsqu'on appelle cette fonction directement, le dessin sera réalisé à l'écran. Par contre, si l'on passe le nom de cette fonction à la fonction *printerPlot()*, le graphique sera envoyé vers l'imprimante au lieu de s'afficher à l'écran.

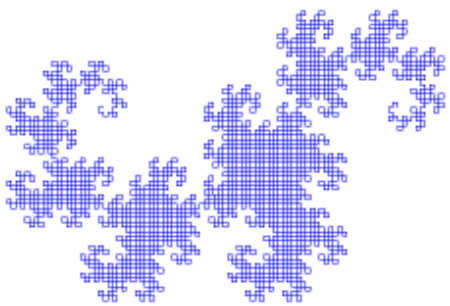
L'appel de *printerPlot()* va ouvrir une boîte de dialogue permettant de régler différentes options d'impression, en particulier le choix de l'imprimante. Il est souvent possible d'imprimer sur des imprimantes virtuelles permettant de créer des fichiers graphiques vectoriels en haute résolution (par exemple au format TIFF ou EPS).



CONCEPTS DE PROGRAMMATION: *Graphiques en haute résolution*

UN DRAGON QUI-NE-CRACHE-PAS-DE-FEU

Pour montrer un exemple d'impression en haute résolution, demandons à la tortue de dessiner une figure complexe nommée « courbe du dragon ». Bien qu'il soit possible de réaliser cette courbe par pliage de la feuille, il est plus facile de recourir à l'ordinateur. L'implémentation de l'algorithme de pliage est toutefois loin d'être trivial [**plus...**].



Puisque l'on s'intéresse à l'impression, la fonction *figure(s, n, flag)* permettant de dessiner la courbe est donnée sans grandes explications. On remarquera néanmoins que la courbe est définie de manière récursive. Les deux appels récursifs différents diminuent tous deux la valeur du paramètre de *n* à *n-1* et l'ancrage de la récursion survient lorsque *n* est nul. De plus, la fonction demande un paramètre *flag* prenant les valeurs *1* ou *-1* qui indique la direction de dessin.

Pour imprimer l'image, on indique les instructions nécessaires dans la fonction *doIt()* qui ne prend pas de paramètre. Si l'on appelle directement la fonction *doIt()*, le dessin apparait à l'écran mais lorsqu'on passe le nom de la fonction *doIt* à la fonction *printerPlot()*, le graphique est imprimé sans faire apparaître la tortue.

```
from gturtle import *
import math

nbGenerations = 12

def doIt():
    rt(90)
    figure(300, nbGenerations, 1)

def figure(s, n, flag):
    if n == 0:
        fd(s);
    else:
        alpha = 45
        if flag == 1:
            alpha = -alpha
            flag = -flag
        lt(alpha)
        figure(s / math.sqrt(2), n - 1, -flag)
        rt(2 * alpha)
        figure(s / math.sqrt(2), n - 1, flag)
        lt(alpha)

makeTurtle()
ht()
setPos(-100, 100) # screen
doIt()
setPos(100, 0)      # printer
printerPlot(doIt)
```

Sélectionner tout le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Il faut faire attention à placer le dessin de manière appropriée sur la feuille avec la fonction *setPos()*. La position de départ dépend de la taille de la fenêtre tortue ainsi que des paramètres d'impression choisis. Il est possible de réaliser l'impression avec un facteur d'agrandissement:math:`k > 1` ou un facteur de réduction :math:`k<1` en tant que second paramètre de la fonction *printerPlot(doIt, k)*.

EXERCICES

1. Joshua Goldstein propose d'utiliser des paires d'instructions mouvement/rotation pour créer de jolies images. Une itération est donc constituée des instructions
   forward(s)
   right(a)

   Dessiner à l'écran puis imprimer les figures de Goldstein suivantes:
   a. 31 itérations avec *s = 300* et *a = 151*
   b. 142 itérations avec *s = 400* et *a = 159.72*

   À vous de trouver la bonne position de départ pour que le graphique tienne sur la feuille!

2. Une étape peut également consister de deux paires mouvement/rotation différentes. Dessiner à l'écran puis imprimer la figure de Goldstein suivante:

   37 itérations avec *s = 77*, *a = 140.86* puis *s = 310* et *a = 112*

3. Dessiner à l'écran puis imprimer la figure de Goldstein suivante fabriquées sur trois paires de mouvement/rotation différentes:

   47 itérations avec *s = 15.4*, *a = 140.86*, puis *s = 62* et *a = 112* puis finalement *s= 57.2* et *a = 130*

Documentation des graphiques tortue

Module import: from gturtle import *

Function	Action
makeTurtle()	Crée une tortue (globale) dans une nouvelle fenêtre et définit toutes les commandes globales
makeTurtle(color)	Idem, mais en spécifiant la couleur de la tortue à créer
makeTurtle("sprites/turtle.gif")	Idem, mais en créant une tortue représentées par l'image de sprite spécifiée
t = Turtle()	Créer un objet tortue *t*
tf = TurtleFrame()	Crée une fenêtre de graphique (*TurtleFrame*) pouvant contenir plusieurs tortues
t = Turtle(tf)	Crée une tortue *t* qui se trouve dans le *TurtleFrame tf*
clone()	Crée un clone de la tortue globale possédant les mêmes propriétés (même couleur, position et orientation)
isDisposed()	Retourne *True* si la fenêtre est fermée
putSleep()	Met l'exécution du programme en pause jusqu'à l'appel de la fonction *wakeUp()*
wakeUp()	Reprend l'exécution d'un programme mis en pause avec *putSleep()*
enableRepaint(False)	Déclenche le rendu automatique de l'écran
repaint()	Restitue l'écran manuellement (après déclenchement du rendu automatique)
savePlayground(fileName, format)	Copie le playground dans un fichier image (format: "png" ou "gif"). Retourne False dans le cas ,ch,ant

Movements

back(distance), bk(distance)	Déplace la tortue en arrière sur la distance indiquée (coordonnées tortue)
forward(distance), fd(distance)	Déplace la tortue en avant sur la distance indiquée (coordonnées tortue)
hideTurtle(), ht()	Cache la tortue, ce qui a pour effet d'accélérer le traçage du dessin
home()	Replace la tortue dans sa position d'origine au centre de la fenêtre et orientée vers le haut
left(angle), lt(angle)	Tourne la tortue de *angle* degrés vers la gauche
penDown(), pd()	Active le crayon (la tortue trace son chemin)
penErase(), pe()	Assigne au crayon la même couleur que le fond d'écran (trace invisible)
leftArc(radius, angle)	Déplace la tortue sur un arc de cercle orienté vers la gauche, d'angle *angle* (en degrés) et de rayon *radius*
leftCircle(radius)	Déplace la tortue sur un cercle de rayon *radius* en partant vers la gauche.

penUp(), pu()	Désactive le crayon (la trace de la tortue devient invisible)
penWidth(width)	Règle la largeur de la trace (en pixels)
right(angle), rt(angle)	Tourne la tortue de *angle* degrés vers la droite
rightArc(radius, angle)	Déplace la tortue sur un arc de cercle orienté vers la droite, d'angle *angle* (en degrés) et de rayon *radius*
rightCircle(radius)	Déplace la tortue sur un cercle de rayon *radius* en partant vers la droite.
setCustomCursor(cursorImage)	Règle le fichier image utilisé en guise de curseur de la souris
setCustomCursor(cursorImage, Point(x, y))	Idem, en indiquant les coordonnées relatives de l'image par rapport au point d'action de la souris.
setLineWidth(width)	Règle la largeur du crayon (en pixels)
showTurtle(), st()	Affiche la tortue globale
speed(speed)	Règle la vitesse du mouvement de la tortue
delay(time)	Arrête l'exécution du programme durant l'intervalle de temps *time* (en millisecondes)
wrap()	Les positions des tortues se trouvant en dehors de l'écran sont envoyées à l'intérieur de la fenêtre par une symétrie sur un tore. Une tortue qui dépasse à gauche réapparait donc à droite et une tortue qui dépasse vers le haut réapparait en bas de l'écran.
clip()	Contraire de *wrap()* : les tortues qui sortent de la fenêtre sont invisibles
getPlaygroundWidth()	Retourne la largueur m du territoire de la tortue (coordonnées -m/2...m/2)
getPlaygroundHeight()	Retourne la hauteur m du territoire de la tortue (coordonnées -m/2...m/2)

Localisation

direction(x, y)	Retourne l'angle pour tourner dans la direction de la position (x, y)
direction(coords)	Idem, en spécifiant les coordonnées coords sous forme de tuple (x,y), de liste [x,y], ou de nombre complexe x + yj
direction(turtle)	Retourne l'angle pour turner dans la direction d'une autre tortue
distance(x, y)	Retourne la distance séparant la position de la tortue et le point de coordonnées (x, y)
distance(coords)	Idem, en spécifiant les coordonnées *coords* sous forme de tuple *(x,y)*, de liste *[x,y]*, ou de nombre complexe *x + yj*
distance(turtle)	Retourne la distance séparant la position de la tortue de la position d'une autre tortue
getPos()	Retourne la position courante de la tortue sous forme de liste
getX()	Retourne la coordonnée *x* de la tortue
getY()	Retourne la coordonnée *y* de la tortue

heading()	Retourne l'orientation actuelle de la tortue en degrés. L'orientation vers le nord correspond à un angle de 0°. Les angles sont orientés dans le sens des aiguilles de la montre.
heading(degrees)	Règle l'orientation de la tortue en degrés. L'orientation vers le nord correspond à un angle de 0°. Les angles sont orientés dans le sens des aiguilles de la montre.
moveTo(x, y)	Déplace la tortue à la position (x,y) en dessinant la trace
moveTo(coords)	Idem, en spécifiant les coordonnées *coords* sous forme de tuple *(x,y)*, de liste *[x,y]*, ou de nombre complexe *x + yj*
setHeading(degrees), setH(degrees)	Règle l'orientation de la tortue (en degrés, 0° correspond au Nord, angles orientés dans le sens des aiguilles de la montre)
setRandomHeading()	Règle l'orientation de manière aléatoire avec un angle de vue compris entre 0° et 360°
setPos(x, y)	Positionne la tortue au point de coordonnées (x,y) sans dessiner la trace
setPos(coords)	Idem, en spécifiant les coordonnées *coords* sous forme de tuple *(x,y)*, de liste *[x,y]*, ou de nombre complexe *x + yj*
setX(x)	Règle la coordonnée *x* de la tortue
setY(y)	Règle la coordonnée *y* de la tortue
setRandomPos(width, height)	Place la tortue sur une position aléatoire située à l'intérieur du rectangle centré à la position actuelle, de largeur *width* et de hauteur *height*
setScreenPos(x, y)	Positionne la tortue globale aux coordonnées d'écran *x* et *y*
setScreenPos(Point(x, y))	Idem en spécifiant les coordonnées par un objet *Point(x,y)*
towards(x, y)	Retourne l'angle d'orientation (en degrés) qu'il faut donner à la tortue pour qu'elle vise le point de coordonnées (x, y).
towards(coords)	Idem, en spécifiant les coordonnées *coords* sous forme de tuple *(x,y)*, de liste *[x,y]*, ou de nombre complexe *x + yj*
towards(turtle)	Retourne l'angle d'orientation (en degrés) qu'il faut donner à la tortue pour qu'elle vise la tortue *turtle*
toTurtlePos(x, y)	Retourne une liste des coordonneés de la tortue globale au point x, y de l'écran
toTurtlePos(Point(x, y))	Idem en spécifiant les coordonnées par un objet *Point(x,y)*
pushState()	Sauvegarde l'état actuel de la tortue sur une pile (Last In, First Out)
popState()	Restaure le dernier état sauvegardé avec *pushState()*. Supprime ce dernier de la pile des états
clearStates()	Vide la pile de sauvegarde des états en supprimant tous les états sauvegardés

Couleurs

askColor(title, defaultColor)	Ouvre une boîte de dialogue de sélection de couleur et retourne la couleur sélectionnée. Si l'utilisateur clique sur le bouton annuler, retourne *None*.

clear()	Efface toutes les traces et cache toutes les tortues en les laissant par contre à leur place
clear(color)	Efface les traces et cache toutes les tortues puis colorie le fond d'écran avec la couleur *color*
clean()	Efface toutes les traces et toutes les tortues. Réinitialise la position de toutes les tortues au centre de l'écran.
clean(color)	Efface toutes les traces et toutes les tortues en coloriant le fond de la fenêtre avec la couleur *color*. Réinitialise la position de toutes les tortues au centre de l'écran.
clearScreen(), cs()	Efface toutes les traces et replace la tortue dans sa position d'origine
dot(diameter)	Dessiner un disque plein de diamètre *diameter* en le remplissant de la couleur assignée au crayon de la tortue.
openDot(diameter)	Dessiner un cercle vide de diamètre *diameter* en coloriant son périmètre de la couleur assignée au crayon de la tortue.
fill()	Remplit la surface fermée qui entoure le point auquel se trouve la tortue avec la couleur de remplissage spécifiée au préalable avec *setFillColor(color)*.
fill(x , y)	Remplit la surface fermée qui entoure le point de coordonnées (x,y) avec la couleur de remplissage spécifiée au préalable avec *setFillColor(color)*
fill(coords)	Idem, en spécifiant les coordonnées *coords* sous forme de tuple *(x,y)*, de liste *[x,y]*, ou de nombre complexe *x + yj*
fillToPoint()	Remplissage continu depuis la position actuelle de la tortue
fillToPoint(x , y)	Remplissage continu depuis le point de coordonnées *(x, y)*
fillToPoint(coords)	Idem, en spécifiant les coordonnées *coords* sous forme de tuple *(x,y)*, de liste *[x,y]*, ou de nombre complexe *x + yj*
fillToHorizontal(y)	Remplissage continu de la surface délimitée par la trace actuelle de la tortue et la droite horizontale formée par les points d'ordonnée *y*
fillToVertical(x)	Remplissage continu de la surface délimitée par la trace actuelle de la tortue et la droite verticale formée par les points d'abscisse *x*
fillOff()	Termine le mode remplissage
getColor()	Retourne la couleur de la tortue globale
getColorStr()	Retourne la couleur de la tortue globale sous la forme d'une chaine de caractères X11
getFillColor()	Retourne la couleur de remplissage actuelle
getFillColorStr()	Retourne la couleur de remplissage actuelle sous la forme d'une chaine de caractères X11
getPixelColor()	Retourne la couleur du pixel (trace ou fond) situé à la position actuelle de la tortue globale
getPixelColorStr()	Idem, en retournant la couleur sous forme de chaine X11
getRandomX11Color()	Retourne une couleur aléatoire sous forme de chaine de caractères X11

makeColor()	Retourne une couleur de référence correspondant à la valeur passée. Exemples de valeurs : ("7FFED4"), ("Aqua-Marine"), (0x7FFED4), (8388564), (0.5, 1.0, 0.83), (128, 255, 212), ("rainbow", n) avec n = 0..1 qui indique une couleur de l'arc-en-ciel : n=0.1 correspond au violet et n=0.9 au rouge
setColor(color)	Règle la couleur de la tortue à *color*.
setPenColor(color)	Règle la couleur du crayon utilisé par la tortue à *color*
setFillColor(color)	Règle la couleur de remplissage utilisée par la tortue à *color*
startPath()	Commence à tenir compte du mouvement de la tortue pour les opérations de remplissage subséquentes.
fillPath()	Contraire de *startPath()* : termine l'opération de remplissage à la position courante de la tortue et remplit la zone ainsi délimitée par la couleur de remplissage.
stampTurtle()	Crée une image de la tortue à la position courante.
stampTurtle(color)	Crée une image de la tortue à la position courante avec la couleur *color*

Fonctions de rappel (callbacks)

makeTurtle(mouseNNN = onMouseNNN) use a comma separator to register more than one	Enregistre la fonction de rappel (callback) *onMouseNNN(x,y)* qui est appelée lorsqu'un événement de la souris survient. Les valeurs possibles pour *NNN* sont : *Pressed*, *Released*, *Clicked*, *Dragged*, *Moved*, *Entered*, *Exited*, *SingleClicked*, *DoubleClicked*, *Hit* : L'invocation se fait dans un thread (fil d'exécution) séparé, *HitX*: idem, mais les événements sont ignorés jusqu'à ce que le traitement de l'événement en cours soit terminé
isLeftMouseButton(), isRightMouseButton()	Retourne vrai si l'événement est généré par le bouton gauche, respectivement droit
makeTurtle(keyNNN = onKeyNNN)	Enregistre la fonction de rappel (callback) *onKeyNNN(keyCode)* qui est appelée lorsqu'une touche du clavier est enfoncée. Les valeurs possibles pour *NNN* sont : *Pressed*, *Hit* : invocation du gestionnaire d'événement dans un thread séparé et *HitX* : idem que *Hit*, mais les événements sont ignorés jusqu'à ce que la dernière exécution de la fonction de rappel ait retourné. *keyCode* est un entier unique qui identifie la touche pressée
getKeyModifiers()	Retourne un nombre entier représentant les touches spéciales du clavier enfoncées (Shift, Ctrl, etc…) et leurs combinaisons
makeTurtle(closeClicked = onCloseClicked)	Enregistre la fonction de rappel *onCloseClicked()* qui est appelée lorsque le bouton *fermer* de la barre de titre est actionné. La fenêtre doit ensuite être fermée manuellement en appelant *dispose()*
makeTurtle(turtleHit = onTurtleHit)	Enregistre la fonction de rappel *onTurtleHit(x, y)* qui est appelée lorsqu'on clique sur l'image de la tortue globale
t = Turtle(turtleHit = onTurtleHit)	Enregistre la fonction de rappel *onTurtleHit(t, x, y)* qui est appelée lorsqu'on clique sur l'image de l'objet tortue *t*

Texte, images et son

addStatusBar(20)	Ajout une barre d'état de 20 pixels de hauteur
beep()	Émet un court son

playTone(freq)	Joue un son de fréquence *freq* (en Hz) et de durée 1000 ms. Fonction bloquante
playTone(freq, block=False)	Idem, mais en version non bloquante, ce qui permet de jouer plusieurs sons à la fois
playTone(freq, duration)	Joue le son de fréquence *freq*` et de durée *duration* (en ms)
playTone([f1, f2, ...])	Joue plusieurs sons consécutifs de fréquences *f1*, *f2*, … et de durée 1000 ms
playTone([(f1, d1), (f2, d2), ...])	Joue plusieurs tons consécutifs indiqués par des tuples (fréquence, durée)
playTone([("c",700), ("e",1500), ...])	Joue plusieurs tons consécutifs en utilisant le nom de la note (A=la, H=si, C=do, D=ré, …, et de durée indiquée en [ms]. La tessiture du générateur de son s'étend du do1, do#1, … jusqu'au si5
playTone([("c",700), ("e",1500), ...], instrument = "piano")	Idem, mais en choisissant le type d'instrument à utiliser pour la restitution du son. Les instruments disponibles sont *piano*, *guitar*, *harp*, *trumpet*, *xylophone*, *organ*, *violin*, *panflute*, *bird*, *seashore* ... (voir la spécification MIDI)
playTone([("c",700), ("e",1500), ...], instrument = "piano", volume=10)	Idem, en rajoutant le choix du volume compris entre 0..100
label(text)	Affiche le texte *text* à la position courante de la tortue
printerPlot(draw)	Envoie vers l'imprimante les commandes de dessins contenues dans la fonction *draw()* dont la référence est passée en argument
setFont(Font font)	Définit la police utilisée par la fonction *label()*
setFontSize(size)	Définit la taille de police utilisée par la fonction *label()*
getTextHeight()	Retourne la hauteur des caractères de la police actuellement en usage (en pixels)
getTextAscent()	Retourne la hauteur du jambage supérieur de la police actuellement en usage (en pixels). Pour plus de détails, consulter
getTextDescent()	Retourne la hauteur du jambage inférieur de la police actuellement en usage (en pixels). Cf. *getTextAscent()*.
getTextWidth(text)	Retourne la largeur en pixels du texte *text* avec la police actuellement en usage.
setStatusText(text)	Affiche le texte *text* dans la barre d'état. Tout texte présent dans la barre d'état au moment de l'appel est écrasé.
setTitel(title)	Affiche la chaine de caractères *title* dans la barre de titre de la fenêtre
img = getImage(path)	Charge et retourne l'image par un fichier JAR, par un fichier du disque local ou par un serveur internet
drawImage(img)	Dessine l'image dans le fond d'écran avec le centre à la position actuelle de la tortue et orienté par l'angle actuelle de la tortue
drawImage(path)	Charge l'image par un fichier JAR, par un fichier du disque local ou par un serveur internet et dessine l'image dans le fond d'écran avec le centre à la position actuelle de la tortue et orienté par l'angle actuelle de la tortue

Boîtes de dialogue **Documentation Entry Dialoque**

msgDlg(message)	Ouvre une boîte de dialogue modale avec le message *message* et un bouton OK
msgDlg(message, title = text)	Idem, en spécifiant le titre de la fenêtre
inputInt(prompt)	Ouvre une boîte de dialogue modale avec les boutons OK/Annuler. OK retourne le nombre entier entré. Un clic sur Annuler ou Fermer termine le programme.
inputInt(prompt, False)	Idem, sauf qu'un clic sur Annuler/Fermer n'interrompt pas le programme mais retourne la valeur *None*
inputFloat(prompt)	Ouvre une boîte de dialogue modale avec les boutons OK/Annuler. OK retourne le nombre à virgule flottante entré. Un clic sur Annuler ou Fermer termine le programme.
inputFloat(prompt, False)	Idem, sauf qu'un clic sur Annuler/Fermer n'interrompt pas le programme mais retourne la valeur *None*
inputString(prompt)	Ouvre une boîte de dialogue modale avec les boutons OK/Annuler. OK retourne la chaine entrée. Un clic sur Annuler ou Fermer termine le programme.
inputString(prompt, False)	Idem, sauf qu'un clic sur Annuler/Fermer n'interrompt pas le programme mais retourne la valeur *None*
input(prompt)	Ouvre une boîte de dialogue modale avec les boutons OK/Annuler. OK retourne le nombre entier, le nombre flottant ou la chaine de caractères saisie par l'utilisateur. Les boutons Annuler ou Fermer terminent le programme.
input(prompt, False)	Idem, sauf qu'un clic sur Annuler/Fermer n'interrompt pas le programme mais retourne la valeur *None*
askYesNo(prompt)	Ouvre une boîte de dialogue modale avec les boutons Oui/Non. Oui retourne *True* et Non retourne *False*. Les boutons Annuler ou Fermer terminent le programme.
askYesNo(prompt, False)	Idem, sauf qu'un clic sur Annuler/Fermer n'interrompt pas le programme mais retourne la valeur *None*

chapter three

2D GRAPHICS & PICTURES

Objectifs d'apprentissage

- Être capable de créer des graphiques 2D simples à l'aide de formes géométriques.
- Être capable d'utiliser le clavier et la souris pour interagir avec la fenêtre de graphiques.
- Être capable d'afficher le graphe de fonctions simples *y=f(x)* dans la fenêtre de graphiques.
- Savoir qu'une image numérisée est constituée de pixels colorés qui peuvent être stockés sous forme de nombres.
- Être capable de charger une image numérique, la modifier par programme de manière spécifique, la représenter à l'écran et la sauver sur un support de stockage.
- Être capable de développer des programmes pilotés par le clavier et la souris.
- Être capable de générer des nombres aléatoires et de les utiliser pour effectuer des expériences impliquant le hasard.
- Être capable d'utiliser les champs de saisie et les menus dans ses propres programmes.

"Une image vaut mieux que mille mots."

Proverbe Ancien

3.1 COORDONNÉES

INTRODUCTION

Nous avons déjà fait nos premières expériences de dessin sur ordinateur avec la tortue. La tortue recèle cependant quelques limitations auxquelles ce chapitre tentera de pallier avec une bibliothèque de création de graphiques plus flexible.

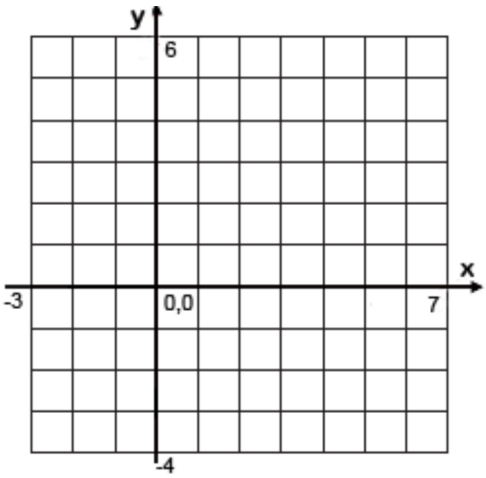
CONCEPTS DE PROGRAMMATION: *Coordinate graphics, Cartesian coordinate system*

OUVRIR LA FENÊTRE DE GRAPHIQUES

La bibliothèque que nous allons utiliser, respectivement la fenêtre de graphiques, s'appellent *GPanel*. Cette bibliothèque de programmation graphique est déjà installée par défaut dans TigerJython mais il est nécessaire de l'importer au début du programme par une instruction *import*. Il est ensuite possible d'utiliser **makeGPanel()** pour créer une nouvelle fenêtre de graphiques.

```
from gpanel import *
makeGPanel(-3, 7, -4, 6)
```

Jusqu'ici, le programme ne fait rien de bien excitant puisqu'il ne fait que d'afficher une fenêtre vide qu'on peut fermer. La fenêtre de graphiques est de forme carrée et utilise un repère cartésien *xOy* que l'on connaît bien des mathématiques :

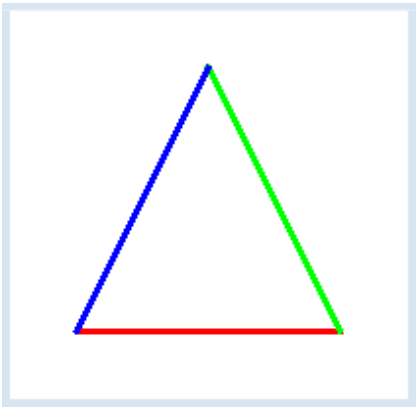


Dans l'exemple ci-dessus, on spécifie les plages des coordonnées *x* et *y* avec les quatre paramètres -3, 7, -4, et 6. Le premier, -3, indique l'abscisse du bord gauche de la fenêtre, 7 l'abscisse de son bord droit, -4 l'ordonnée du bord inférieur et 6 celle de son bord supérieur.

MEMENTO

On peut créer une fenêtre avec **makeGPanel().** On indique la plage du repère cartésien que recouvre la fenêtre avec les quatre paramètres : *makeGPanel(xmin, xmax, ymin, ymax)*. On peut également spécifier le titre de la fenêtre en guise de premier paramètre:
makeGPanel(title, xmin, xmax, ymin, ymax)

DESSINER DES LIGNES



Une fois la fenêtre ouverte, on peut dessiner librement à l'intérieur à l'aide d'une grande panoplie de fonctions. La fonction **line()** permet par exemple de dessiner des lignes et la fonction **setColor()**

permet de changer la couleur utilisée. On peut donc créer le triangle coloré ci-contre avec le programme indiqué plus bas.

Pour chaque ligne à dessiner, on indique les coordonnées cartésiennes du point de départ et du point d'arrivée du segment. Les sommets du triangle devraient avoir les coordonnées suivantes : (1, -1) (5, -1) (3, 3). Bien entendu, on ne voit apparaître un triangle que dans la mesure où le système de coordonnées est choisi judicieusement. Pour obtenir des images non déformées, il est nécessaire que les deux axes utilisent la même longueur par unité. Pour ce faire, il faut que *xmax-xmin = ymax-ymin* puisque la fenêtre est carrée.

```
from gpanel import *

makeGPanel("My window", 0, 6, -2, 4)

lineWidth(3)
setColor("red")
line(1, -1, 5, -1)
setColor("green")
line(5, -1, 3, 3)
setColor("blue")
line(3, 3, 1, -1)
```

MEMENTO

On peut indiquer la largeur des traits en pixels avec la fonction **lineWidth()**.

CERCLES ET RECTANGLES

GPanel n'est pas limité au dessin de lignes : cercles, ellipses, rectangles, triangles et arcs d'ellipses sont également de la partie. GPanel peut même écrire du texte. On peut dessiner un disque plein de rayon *radius* avec la fonction *fillCircle(radius)*. Mais avant de dessiner un disque, il faut positionner le curseur de dessin avec *move(x,y)* pour définir les coordonnées du centre du disque.

fillRectangle(length, width) va dessiner un rectangle de largeur et de longueur indiquées et centré à la position du curseur de dessin. Le programme suivant dessine plusieurs carrés et disque en utilisant une boucle *while*.

```
from gpanel import *

makeGPanel(0, 20, 0, 20)

setColor("red")
x = 2
y = 2
while y < 20:
    move(x, y)
    fillCircle(1)
    move(x, 20 - y)
    fillRectangle(2, 2)
    x = x + 2
    y = y + 2
```

Sélectionner le code source (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

On peut dessiner de nombreuses figures différentes avec GPanel. Voici les fonctions les plus importantes :

point(x, y)	Un point aux coordonnées x, y
line(x1, y1, x2, y2)	Un segment de droite défini par ses extrémités
rectangle(width, height)	Un rectangle (vide)
fillRectangle(width, height)	Un rectangle plein
rectangle(x1, y1, x2, y2)	Un rectangle vide défini par deux sommets
fillRectangle(x1, y1, x2, y2)	Un rectangle plein définir par deux sommets
fillTriangle(x1, y1, ..., y3)	Un triangle défini par ses trois sommets
circle(r)	Un cercle (vide) de rayon *r*
fillCircle(r)	Un disque (plein) de rayon *r*
ellipse(a, b)	Une ellipse d'axes *a* et *b*
fillEllipse(a, b)	Une ellipse pleine
arc(r, a, b)	Un arc d'ellipse de rayon *r* et d'axes *a* et *b*
text("t")	Écrire le texte *t* à la position du cureur
move(x, y)	Modifier la position du curseur de dessin

Pour les cercles, arcs, ellipses, texte et rectangles qui sont uniquement définis par leur longueur et leur largeur, il est nécessaire de définir au préalable leur positionnement en plaçant le curseur de dessin avec la fonction *move()*.

GPanel connaît les ***couleurs X11***. Cette palette comporte quelques dizaines de couleurs auxquelles sont associés es noms précis disponibles sur Internet à l'adresse **http://cng.seas.rochester.edu/CNG/docs/x11color.html**. On peut choisir n'importe laquelle de ces couleurs avec *setColor(color)*.

EXERCISES

1. Dessiner le visage ci-dessous :

2. À quoi un arc-en-ciel ressemble-t-il ? Demander à l'ordinateur d'en dessiner un. Utiliser la fonction *circle(r)* de sorte qu'uniquement la partie supérieure du cercle soit visible.

3.2 BOUCLES FOR

INTRODUCTION

Souvent, il est nécessaire de maintenir un compteur dans une structure de répétition. Pour ceci, il faut une variable au sein du bloc de répétition qui change d'une certaine quantité à chaque itération de la boucle. Il est plus simple et naturel de réaliser ceci avec une boucle *for* qu'avec une boucle *while*. Pour ce faire, il faut tout d'abord apprendre à connaître la fonction *range()*. Dans le cas le plus simple, *range(n)* ne prend qu'un seul paramètre également appelé *stop value* en anglais et retourne une suite de nombres naturels consécutifs débutant à 0 et se terminant à *n-1* compris.
Il est utile de tester cette fonction *range()* dans une session interactive de l'interpréteur. L'appel *range(10)* va par exemple afficher la liste de nombres

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9] dans la fenêtre de sortie. Testez cette fonction avec plusieurs valeurs différentes, parmi lesquelles 0 pour bien comprendre son fonctionnement. Remarquez que la valeur 10 ne figure pas dans la liste de nombres retournés. Ce paramètre 10 indique plutôt le nombre d'éléments présents dans la liste générée.

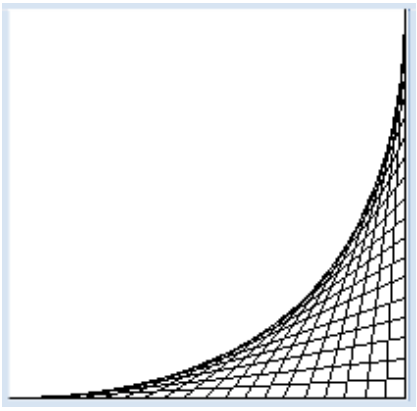
CONCEPTS DE PROGRAMMATION: *Itération, structure for, boucles imbriquées.*

FAMILLE DE COURBES

On peut dessiner une courbe sympathique à l'aide de 20 segments droits à l'aide d'une **boucle for**.

```
from gpanel import *

makeGPanel(0, 20, 0, 20)

for i in range(21):
    line(i, 0, 20, i)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V coller)



MEMENTO

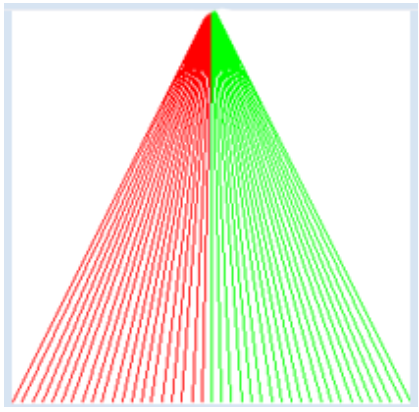
L'instruction **for i in range(n)** parcourt les nombres de *0* à *n-1*, à savoir un total de *n* nombres en tout. Les points de densification des segments droits forment une ***courbe de Bézier quadratique***.

RANGE() AVEC DEUX PARAMÈTRES

La fonction *range* peut également prendre **deux paramètres.**Dans ce cas, le premier indique la valeur de départ de la liste et le second la valeur d'arrêt qui n'est cependant pas incluse dans la liste.

En écrivant par exemple *range(2,9)*, on récupère la liste de nombres *[2, 3, 4, 5, 6, 7, 8]*. Tester la fonction *range* sur d'autres couples de paramètres pour bien comprendre son fonctionnement. Tester entre autres les couples suivants : (0,0), (0,1), (1,2), (1,3) et … (5,3) et observer attentivement les listes retournées.

Le programme suivant dessine des segments droits avec deux couleurs différentes contrôlées par le compteur d'itération *i*. Leur point d'origine se trouve sur l'axe des x (y=0) dont l'abscisse varie entre -20 et 20 par pas de 1. L'extrémité de chaque segment est le point (0, 40).



```
from gpanel import *

makeGPanel(-20, 20, 0, 40)

for i in range(-20, 21):
    if i < 0:
        setColor("red")
    else:
        setColor("green")
    line(i, 0, 0, 40)
```

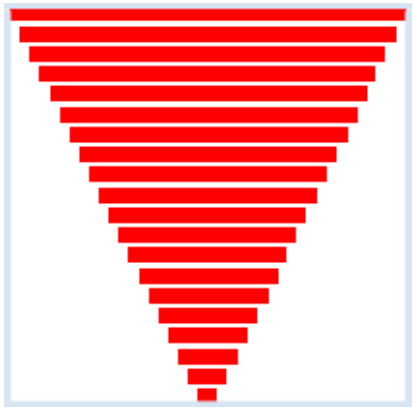
MEMENTO

La boucle **for i in range(start, stop)** avec les entiers *start* et *stop* commence à *i = start* et se termine à *i = stop – 1* en augmentant le compteur d'itérations *i* d'une unité à chaque itération de la boucle. De ce fait, il faut que *start* soit inférieur à *stop* pour que le programme puisse entrer dans la boucle puisque *range(start, stop)* retourne une liste vide si *start >= stop*.

RANGE() AVEC TROIS PARAMÈTRES

On peut même appeler la fonction *range()* avec **trois paramètres.**Dans ce cas, le premier représente la valeur de départ, le deuxième la valeur d'arrêt et le troisième représente l'incrément ajouté à chaque itération au compteur de boucle *i*. Ce troisième paramètre qui vaut 1 par défaut permet donc de modifier la grandeur du pas de chaque itération.

L'instruction *print range(2, 15, 3)* va par exemple afficher la liste de nombres *[2, 5, 8, 11, 14]* dans la fenêtre de sortie du programme.



Dans le graphique ci-contre, on voit une pyramide renversée formée de rectangles pleins. Le plus petit rectangle a une largeur de 2 et le plus grand une largeur de 40. Étudiez attentivement le programme ci-dessous permettant de réaliser cette figure :

```
from gpanel import *

makeGPanel(0, 40, 0, 40)
```

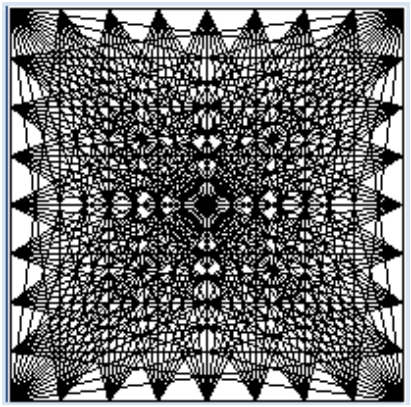
```
setColor("red")
y = 1
for i in range(2, 41, 2):
    move(20, y)
    fillRectangle(i, 1.5)
    y = y + 2
```

MEMENTO

La boucle **for i in range(start, stop, step)**commence avec *i* = *start* et se termine avec une valeur inférieure à *stop*. Le compteur de boucle *i* est incrémenté de *step* à chaque itération de la boucle. Il est également possible de choisir des nombres négatifs pour les paramètres *start*, *stop* et *step*. Si *step* > *0*, il faut que *start* < *stop* tandis que si *step* < *0*, il faut que *start* > *stop*.

En effet, si *step* est négatif, *i* est réduit de *step* à chaque itération ; la dernière valeur est alors supérieure à *stop*.

BOUCLES *FOR* IMBRIQUÉES (Moiré)



Des lignes serrées disposées les unes au-dessus des autres peuvent faire apparaître des effets optiques appelés *Effet de moiré*. Dans un carré, on dessine pour chacun des 10 points équidistants situés sur le côté inférieur une ligne issue de ce point et aboutissant sur chacun des 10 points équidistants du côté supérieur. On fait ensuite de même pour les lignes issues des 10 points équidistants du côté gauche vers les points du côté droit.

Étudier attentivement le programme ci-dessous qui utilise à deux reprises le concept de ***boucles imbriquées***.

```
from gpanel import *

makeGPanel(0, 10, 0, 10)

for i in range(11):
    for k in range (11):
        line(i, 0, k, 10)
        delay(50)
for i in range(11):
    for k in range (11):
        line(0, i, 10, k)
        delay(50)
```

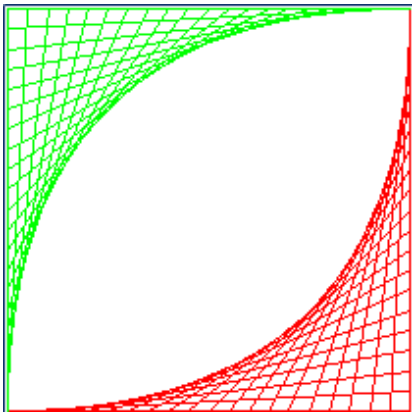
MEMENTO

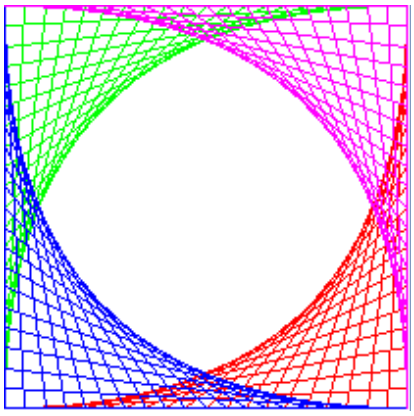
Ce programme n'est certainement pas tout simple à comprendre mais il est d'une importance cruciale. Une clé pour le comprendre est de supposer que le compteur *i* de la boucle extérieure a une valeur constante (initialement 0). La boucle intérieure s'exécute alors avec cette valeur de *i* en faisant varier *k* de *0* à *10* non compris, ce qui a pour effet de dessiner les 10 droites issues du point inférieur gauche du carré et aboutissant sur chacun des points équidistants du côté supérieur. Une fois que cette première boucle intérieure est terminée, la boucle extérieure augmente *i* à 1 pour dessiner les 10 segments issus du point inférieur suivant etc…

La fonction **delay(deltaT)**met le programme en pause durant *deltaT* millisecondes, ce qui permet d'observer la manière dont le motif émerge progressivement.

EXERCICES

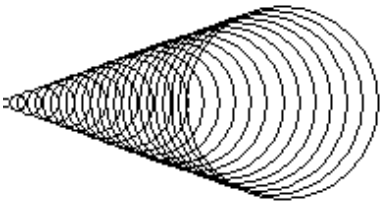
1. On obtiendra un graphique encore plus sympathique que celui obtenu dans l'exemple 1 si l'on utilise des couleurs. Dessiner une seconde famille de courbes avec une couleur différente (Figure a).

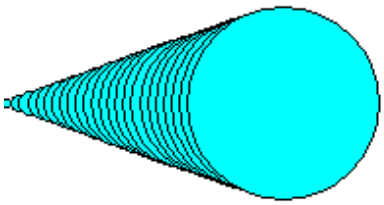
(a) 

(b) 

La famille de courbes bleues (Figure b) est dessinée avec *line(i, 0, 0, 20 - i)*. Seriez-vous capables de dessiner également la famille rose ?

2. Dessiner une famille de cercles vides, puis de disques pleins.





On peut dessiner la famille de cercles coloriés avec la procédure suivante : choisir la couleur cyan, dessiner un disque plein de rayon *y*, choisir ensuite la couleur noire, dessiner un cercle vide de bord noir de même rayon et continuer ainsi de suite :

```
setColor("cyan")
fillCircle(y)
setColor("black")
circle(y)
```

3. Dans l'exemple 3, on a dessiné une pyramide renversée. Dessiner une « vraie » pyramide avec trois couleurs. Utiliser pour ce faire une boucle *for* qui compte les étages pour contrôler le changement de couleur.

3.3 PROGRAMMATION STRUCTURÉE

INTRODUCTION

Le concept de variable est très important en programmation. De ce fait, il faut y prêter une attention particulière et faire un effort spécial pour le comprendre en profondeur. Nous avons déjà vu qu'une variable est une boîte en mémoire référencée par un nom et pouvant contenir des données. On a vu également qu'un paramètre peut être vu comme un emplacement mémoire « volatile » qui reçoit une valeur accessible au sein de la fonction durant son exécution.

CONCEPTS DE PROGRAMMATION: *Constantes, programmation procédurale, réutilisabilité*

UNE MOSAÏQUE DE 10x10 PIERRES

On vous demande de créer une belle mosaïque constituée de pierres de différentes couleurs dont le côté mesure 10. Il faut les disposer côte à côte sur un canevas de taille 400x400. Comme vous êtes paresseux, vous allez faire appel à l'ordinateur pour cette tâche ennuyeuse et hautement répétitive en lui disant de placer les pierres de **couleur aléatoire** ligne par ligne. Utiliser **delay(1)** pour créer une courte pause entre chaque pierre permettant d'observer la construction de la mosaïque.

```
from gpanel import *
makeGPanel(0, 400, 0, 400)

for y in range(0, 400, 10):
    for x in range(0, 400, 10):
        setColor(getRandomX11Color())
        move(x + 5, y + 5)
        fillRectangle(10, 10)
        delay(1)
```

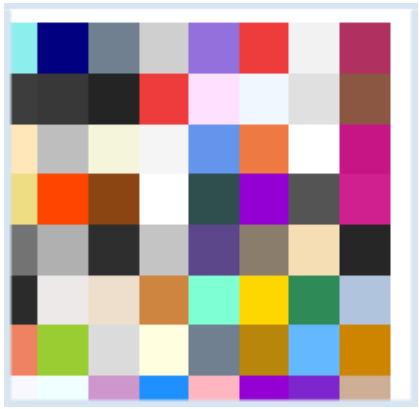
Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

À chaque fois qu'il faut parcourir une grille, deux boucles *for* imbriquées constitueront un outil de choix. Réfléchissez pourquoi les pierres sont disposées ligne par ligne du haut vers le bas. On ajoute 5 pixels vers la droite et vers le haut dans l'appel *move()* parce que le curseur de dessin doit se trouver au centre des rectangles dessinés.

La méthode **getRandomX11Color()** retourne une chaine de caractères représentant l'une des couleurs de la palette X11 que l'on peut ensuite passer à la fonction *setColor()*. On commence donc par appeler la fonction *getRandomX11Color()* et ensuite *setColor()*.

NOMBRES MAGIQUES



Deux semaines plus tard, vous recevez une livraison de pierres qui sont cinq fois plus grandes, avec un côté de 50. L'ordinateur est censé les disposer sur le même canevas de 400x400. Comment faut-il ajuster le programme ? Vous examinez bien entendu le précédent code mais vous êtes incapables de vous rappeler ce que font les différentes lignes de code.

Vous vous dites alors que vous allez remplacer toutes les occurrences de 10 par 50. Et là, mauvaise surprise, vous vous rendez compte que le programme fait n'importe quoi. Maintenant, la mosaïque ne couvre plus l'ensemble du canevas.

Vous vous résolvez alors à parcourir le programme ligne à ligne pour trouver les erreurs. Si un tel travail est nécessaire pour adapter un programme à une nouvelle situation (réutilisation), vous avez écrit votre programme avec beaucoup d'amateurisme même s'il fonctionne correctement. Il faut s'habituer à coder proprement pour développer des programmes flexibles et facilement ajustables aux nouvelles situations.

Comment faut-il faire pour coder proprement ? Au lieu de coder la taille des pierres en dur dans le code, il serait plus judicieux de définir une variable *size* qui peut être utilisée en lieu et place de la taille des pierres dans l'ensemble du programme. Dans le but de structurer encore davantage le programme, on devrait également développer à part une fonction **drawStone()** pour le placement des pierres..

```
from gpanel import *

def drawStone(x, y):
    setColor(getRandomX11Color())
    move(x + size/2, y + size/2)
    fillRectangle(size, size)

makeGPanel(0, 400, 0, 400)

size = 50

for x in range(0, 400, size):
    for y in range(0, 400, size):
        drawStone(x, y)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

De manière générale, l'utilisation de valeurs numériques codées en dur dans le code donne lieu à des programmes peu réutilisables. Il vaut mieux définir des variables et les utiliser à la place des valeurs numériques codées en dur. Pour indiquer que celles-ci ne devraient pas changer, on leur donne un nom entièrement en majuscules et on les appelle alors des **constants**.

Une variable qui est définie dans le programme principal en dehors de toute autre fonction peut également être lue (mais non écrite) de l'intérieur des fonctions. C'est la raison pour laquelle on les appelle **variables globales**.

D'autre part, il est judicieux de placer les bouts de code autonomes dans des fonctions séparées. Cela comporte plusieurs avantages : premièrement, on peut facilement reconnaitre le nom de la fonction dans le code et comprendre ce qu'elle fait pour autant que son nom soit bien choisit et descriptif. Deuxièmement, on peut appeler cette fonction en de nombreuses

situations sans devoir recopier le code dans tous les coins de notre programme. Finalement, le programme devient bien plus lisible et compréhensible. Cette façon de programmer s'appelle **programmation structurée** ou **programmation procédurale** et revêt une importance capitale dans l'art de programmer de manière élégante.

EXERCICES

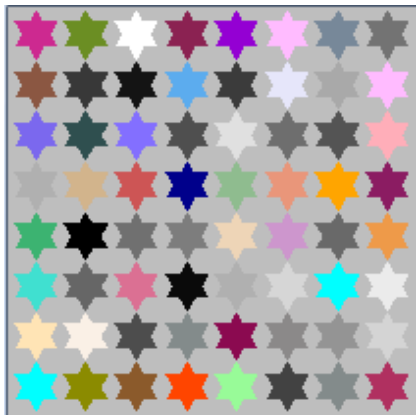
1. Le théorème de Pythagore permet de prouver que l'instruction:

   ```
   fillTriangle(x - math.sqrt(3)/2 * r, y - r/2,
                x + math.sqrt(3)/2 * r, y - r/2,
                x, y + r);
   ```

   dessine un triangle équilatéral dont le cercle circonscrit de rayon *r* est centré en (x, y). Vérifier ceci dans une fenêtre *GPanel* avec un repère cartésien dans la plage -1..1 pour les deux axes. Débrouillez-vous pour que l'instruction ci-dessus dessiner un triangle de cercle circonscrit centré à l'origine et de rayon 1.

2. Réutiliser le code de l'exercice 1 et définir une fonction *star(x, y, r)* qui dessine une étoile formée de deux triangles équilatéraux centrés en (x, y) et de taille (rayon du cercle circonscrit) *r*. Dessiner plusieurs étoiles pour tester la fonction *star()*.

3. Améliorer la fonction *star()* avec un paramètre permettant d'indiquer la couleur de remplissage de l'étoile. Dessiner une mosaïque sur un fond gris composée de 50x50 étoiles. Veiller à garder une taille d'étoiles cohérente et à ce qu'aucune des étoile ne soit remplie avec la même couleur que le fond de la fenêtre.



3.4 FONCTIONS AVEC VALEUR DE RETOUR

INTRODUCTION

Nous avons déjà vu comment définir une fonction avec ou sans paramètre et comment l'appeler. Vous avez sans doute remarqué que le concept de fonction utilisé en programmation diffère sensiblement des fonctions que l'on trouve en mathématiques. En effet, en mathématiques, les fonctions *y = f(x)* possèdent une variable indépendante et pour chaque valeur de *x*, la fonction retourne une valeur de la variable dépendante *y*. Un exemple classique est la fonction quadratique

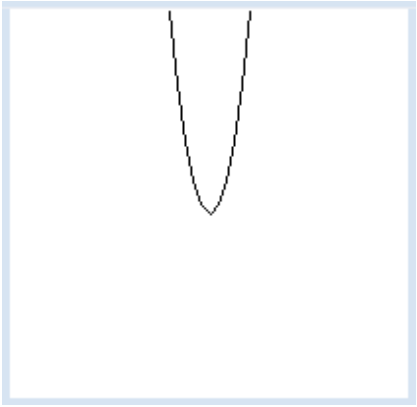
$y = x^2$. qui, à partir des nombres x = 0, 1, 2, 3 retourne les carrés 0, 1, 4, 9.

En Python, on peut également définir des fonctions qui calculent des valeurs et les "retournent" sous forme de variables.

CONCEPTS DE PROGRAMMATION: *Valeur de retour d'une fonction, discrétisation*

LE MOT-CLÉ *RETURN*

On peut définir une fonction *squarenumber(x)* qui calcule le carré x * x pour un nombre donné x, exactement comme en mathématiques. La fonction retourne cette valeur avec le mot-clé return. Cela permet d'esquisser le graphe de la fonction dans GPanel. Pour dessiner le graphe, le mieux est d'utiliser *draw(x, y)* qui dessine des segments droits depuis la dernière position du curseur graphique jusqu'au point (x, y) indiqué en paramètre et qui place ensuite le curseur à la position (x, y). Après l'apparition de la fenêtre GPanel, le curseur graphique se trouve à l'origine (0, 0). Il faut donc ajuster sa position sur le début du graphe avec *move()* sans quoi le segment de départ sera faux.



```
from gpanel import *
makeGPanel(-25, 25, -25, 25)

def squarenumber(x):
    y = x * x
    return y

for x in range(-5, 6):
    y = squarenumber(x)
    if x == -5:
        move(x, y)
    else:
        draw(x, y)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

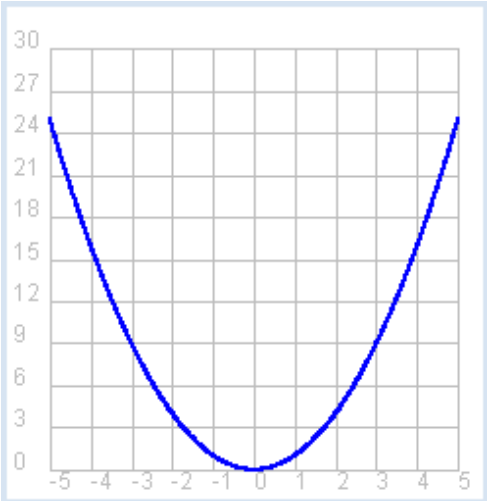
Le mot-clé **return**, permet à une fonction de retourner une valeur au code appelant et d'interrompre son exécution. Si *return* est placé avant la fin du bloc, la fonction s'interrompt au et n'exécute pas les instructions qui se trouvent en-dessous du *return*.

Cependant, comme vous l'avez remarqué, il existe également en informatique des fonctions qui ne retournent pas de valeur et qui produisent tout de même un effet sur l'ensemble du programme. Les fonctions sont même capables de cumuler les deux : retourner une valeur et produire un effet sur l'ensemble du programme, indépendamment de la valeur de retour [**plus...**].

Cette représentation graphique de la fonction quadratique n'est pas encore satisfaisante. En plus de l'absence de système de coordonnées, le graphe tracé est bien trop anguleux. Cela vient du fait que l'on a évalué la fonction en un nombre trop restreint de points entiers que l'on a ensuite connectés par des segments droits. Cela montre une faiblesse essentielle de l'informatique par rapport aux mathématiques : bien que notre fonction livre pour chaque abscisse *x* une image *y*, on ne peut l'évaluer qu'un en nombre fini de points. On dit que l'axe *Ox* continu est **discrétisé** ou **réparti en points discrets**.

NOMBRES DÉCIMAUX (FLOATS)

On peut améliorer cette représentation en choisissant de calculer des points plus proches les uns des autres sur l'axe des *x*. Par exemple, on peut parcourir l'ensemble des abscisses entre -5 et 5 en faisant des centaines de pas. On peut même dessiner un repère cartésien.



Malheureusement, la fonction *range* utilisée pour contrôler la boucle *for* n'accepte que des valeurs entières pour la taille des pas. Pour obtenir une meilleure résolution, il faut donc utiliser une boucle *while*. De cette manière, on peut très bien incrémenter l'abscisse *x* de 0.01 à chaque pas. Avec ce changement, *x* n'est plus considéré comme un nombre entier par *Python*, mais comme un nombre décimal (float).

```
from gpanel import *
makeGPanel(-6, 6, -3, 33)
setColor("gray")
drawGrid(-5, 5, 0, 30)

def squarenumber(x):
    y = x * x
    return y

setColor("blue")
lineWidth(2)
x  = -5
while x < 5:
    y = squarenumber(x)
    if x == -5:
        move(x, y)
    else:
        draw(x, y)
    x = x + 0.01
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

En *Python*, les nombres décimaux sont appelées des flottants (**float**). *A contrario* des mathématiques, les nombres décimaux ne comptent tous qu'un nombre fini de décimales en informatique. En Python, les flottants sont stockés avec 14 décimales. Dans d'autres langages de programmation, de tels nombres sont appelés **double**. Un exemple courant est qu'un ordinateur ne pourra jamais vraiment stocker le nombre π dans sa mémoire puisque ce dernier est irrationnel et comporte de ce fait une infinité de décimales sans période. Il faut donc se résoudre à réaliser des calculs avec une précision limitée à 14 décimales au maximum.

Pour dessiner un repère cartésien, procéder de la manière suivante :

- Étendre le système de coordonnées d'environ 10% horizontalement et verticalement pour permettre l'affichage des étiquettes d'axes. Ainsi, on passera à l'intervalle [-6, 6] pour l'axe *x* et à [-3, 33] pour l'axe *y*.
- Appeler *drawGrid()* avec les coordonnées que l'on avait l'intention d'utiliser en guise de paramètres. Cela engendre une grille de 10x10 carrés.

EXERCICES

1. Définir la fonction *moyenne(a, b)* qui retourne la moyenne arithmétique des deux paramètres. Tester la fonction dans la console.

2. Examiner le comportement de la fonction y = cos(x). Comment diffère-t-elle de la fonction y = sin(x)?

3. Afficher le graphe de la fonction *y = sin(5x)* dans une fenêtre GPanel pour des valeurs de l'axe des *x* comprises entre 0 et 2π avec une résolution de 0.01. On peut obtenir la valeur de π avec l'identifiant *pi* disponible dans le module *math*. Refaire le graphe avec un coefficient de *x* différent de 5 au sein de la fonction *sin*. Quelle est l'effet de ce coefficient sur le graphe de la sinusoïde engendrée ?

4. Définir la fonction *f(x) = 1 / sin(x)* pour la représenter dans une fenêtre GPanel sur l'intervalle [-5 ;5] (pour les deux axes) avec une résolution de 0.001. Dessiner le système de coordonnées dans une couleur différente. Relever les caractéristiques intéressantes du graphe.

3.5 VARIABLES GLOBALES ET ANIMATIONS

INTRODUCTION

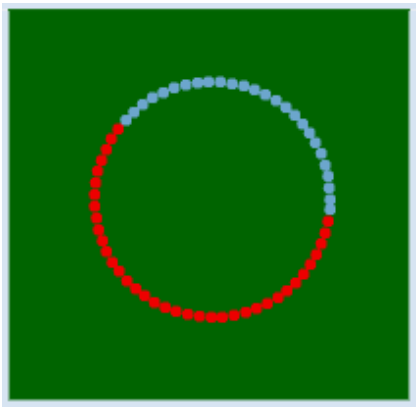
Les graphiques par ordinateur sont souvent utilisés pour représenter des données variant avec le temps. On s'en sert par exemple pour simuler un processus physique ou biologique ou encore pour créer des jeux. On parle alors généralement **d'animation**. Pour montrer une séquence temporelle, de nouvelles images sont dessinées les unes après les autres, espacées de manière régulière, ce que l'on nomme **pas d'animation**.

CONCEPTS DE PROGRAMMATION: *Variables globales, effets de bord, double buffering*

MODIFIER LES VARIABLES GLOBALES

On aimerait illustrer une balle qui se déplace sur un cercle. On obtient un mouvement circulaire de rayon 1 en calculant la coordonnée $x = cos(t)$ et $y = sin(t)$ où t est un paramètre croissant correspond au temps qui s'écoule. Si l'on désire un autre rayon que 1, il faut multiplier chacune des coordonnées par le rayon désiré.

La fonction **step()** dessine la situation pour chaque pas de l'animation. Une fois que la balle a parcouru le cercle entièrement, on voudrait changer sa couleur.



Il est très courant d'introduire une **boucle d'animation** infinie dans le programme principal dont le rôle est d'appeler la fonction *step()* de manière répétitive. En terminant cette boucle par un délai, on peut définir les pauses à marquer entre chaque étape de l'animation et de ce fait régler la vitesse d'animation. La fonction *step()* incrémente la **variable globale** tà chaque pas de la simulation et la réinitialise à 0 une fois le tour complet effectué. On sait que le tour est terminé lorsque $t >= 2*pi = 2 * 3.14 = 6.28$. Une fois le tour terminé, on change de couleur.

```
import math
from gpanel import *

def step():
    global t
    x = r * math.cos(t)
    y = r * math.sin(t)
    t = t + 0.1
    if t > 6.28:
        t = 0
        setColor(getRandomX11Color())
    move(x, y)
    fillCircle(10)

makeGPanel(-500, 500, -500, 500)
bgColor("darkgreen")

t = 0
r = 200
```

```
while True:
    step()
    delay(10)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Python ne permet pas de changer la valeur des variables globales de l'intérieur d'une fonction, car cela pourrait poser de nombreux problèmes qu'il faut éviter à tout prix. On peut cependant faire sauter cette sécurité en déclarant la variable comme globale au sein de la fonction à l'aide du mot-clé **global**.

Ce mot-clé *global* doit être utilisé avec la plus grande prudence car il implique un risque dont il faut être conscient : toute fonction peut non seulement modifier une variable déclarée comme globale, mais il peut également la créer comme le montre l'exemple suivant ::

```
def set():
    global a
    a = 2

def get():
    print "a =", a
set()
get()
```

Du fait que *set()* engendre une variable *a* qui est visible dans tout le programme, on dit que la fonction *set()* engendre des **effets de bord**.

Remarquez en passant la manière dont la commande *print* permet d'écrire plusieurs choses différentes dans la fenêtre de sortie en les séparant par des virgules. Dans la sortie, les virgules sont remplacées par des espaces.

ANIMATIONS FLUIDES

Pour que l'animation soit fluide et régulière, il faut s'arranger pour que les pauses entre chaque étape de l'animation soient de même durée, sans quoi le mouvement risque d'apparaître saccadé. La fonction **step()** permet de préparer la prochaine étape de l'animation. Le temps d'exécution de cette opération peut varier selon la situation, du fait que le programme n'exécute pas toujours les mêmes parties du code d'une part et surtout parce que le processeur peut être plus ou moins occupé à d'autres tâches de fond qui pourraient retarder l'exécution du code Python à certains moments. Pour compenser cette variation dans la durée d'exécution de *step()*, on peut utiliser l'astuce suivante à laquelle vous avez peut-être déjà pensé : avant **d'appeler step()**, on mémorise le temps actuel dans la variable *startTime*. Après avoir terminé l'exécution de *step()*, on attend dans une boucle d'attente jusqu'à ce que la **difference** entre l'heure actuelle et l'heure précédemment mémorisée corresponde à la période d'attente désirée entre chaque étape de l'animation.

Le programme suivant anime un ballon de football se déplaçant d'un but à l'autre. Pour ce faire, il utilise l'image **football.gif** situé dans le dossier *sprites* de l'archive *TigerJython*. Vous pouvez naturellement utiliser une image personnalisée en la sauvant dans le dossier approprié de votre ordinateur et en passant en tant que paramètre de la fonction *image()* son chemin absolu ou relatif par rapport à l'emplacement de votre programme Python.

```
from gpanel import *
import time

def step():
    global x
    global v
    clear()
    lineWidth(5)
    move(25, 300)
    rectangle(50, 100)
    move(575, 300)
    rectangle(50, 100)
    x = x + v
    image("_sprites/football.gif", x, 275)
    if x > 500 or x < 50:
        v = -v

makeGPanel(0, 600, 0, 600)
bgColor("forestgreen")
enableRepaint(False)

x = 300
v = 10

while True:
    startTime = time.clock()
    step()
    repaint()
    while (time.clock() - startTime)  < 0.020:
        delay(1)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Avec *time.clock()*, on peut récupérer l'heure actuelle sous forme d'un nombre décimal. La valeur retournée est dépendante de l'ordinateur et du système d'exploitation. Dans certains environnements, *time.clock()* retourne le temps processeur et dans d'autres le temps écoulé depuis le premier appel à *clock()*. Mais comme on ne s'intéresse qu'à la différence entre les deux mesures du temps, cela n'a pas d'importance. Il suffit de mémoriser l'heure avant l'appel de *step()* et d'attendre à la fin de boucle d'animation, avec *delay(1)*, que la différence entre l'heure actuelle et celle mémorisée dans *startTime* corresponde environ à la période d'animation désirée. Notez cette astuce car vous allez devoir la réutiliser à chaque fois qu'il faudra faire quelque chose à intervalles réguliers.

Chaque commande graphique est immédiatement visible dans la fenêtre GPanel. De ce fait, effacer avec **clear()** pendant une animation va brièvement faire apparaître une fenêtre vide, ce qui cause un clignotement de l'animation désagréable. Pour éviter ce phénomène, il faut utiliser la technique du **double buffering** lors du rendu de nos animations.

Cette technique consiste à utiliser l'instruction **enableRepaint(False)**, qui permet d'effectuer le rendu graphique dans une mémoire tampon (buffer en anglais) hors écran et de n'afficher l'image que lorsque le rendu est terminé, ce qui réduit considérablement les effets de clignotement indésirables. Ainsi, la fonction *clear()* n'affecte que le tampon du rendu graphique et n'efface plus le contenu de la fenêtre graphique. Il faut par contre prendre soin d'afficher manuellement le contenu du tampon de rendu sur l'écran au moment opportun en appelant **repaint()**.

Il faut veiller également à déclarer les variables *x* et *v* comme globales au sein de la fonction *step()* puisqu'il s'agit de variables globales modifiées depuis l'intérieur de la fonction.

EXERCICES

1. Si l'on ne déplace pas les coordonnées *x* et *y* en utilisant les fonctions sinus et cosinus ordinaires, comme dans le premier programme, mais avec des variations de *t* différentes, cela va engendrer des courbes intéressantes appelées **courbes de Lissajoux**. Dessiner de telles courbes avec une résolution de 1/1000 pour *t* dans l'intervalle [0, 2π] et avec les coordonnées

   x = cos(4.5 * t) und y = sin(6.3 * t)

2. Au lieu d'utiliser des valeurs fixes pour le coefficient de *t*, utiliser des variables *omega_x* et *omega_y* prenant les valeurs suivantes :

omega_x	omega_y
3	5
3	7
5	7

   Y a-t-il un lien entre les figures dessinées et les valeurs prises par *omega_x* et *omega_y* ?

3. Dessiner, dans un GPanel dont les deux axes s'étendent entre -2 et 2, les courbes de Lissajoux suivantes : prendre *omega_x = 2* et *omega_y = 7*, pour *t* dans l'intervalle [0, 2π], avec une résolution de 1/100. Au lieu de relier les points de la courbe avec des segments droits, dessiner en chaque point de l'animation un disque de rayon 0.2. On obtient alors une image qui fait penser à un slinky. Comme le montre l'illustration ci-dessous, on peut utiliser des couleurs aléatoires pour chaque disque avec *getRandomX11Color().* N'hésitez pas à vous amuser avec cet exercice !

3.6 CONTRÔLE DU PROGRAMME AU CLAVIER

INTRODUCTION

Les programmes gagnent en interactivité lorsque l'utilisateur peut contrôler son exécution avec les touches du clavier. Bien que les frappes sur les touches soient en fait des événements survenant indépendamment de l'exécution du programme, elles peuvent être capturées dans le GPanel à l'aide de fonctions de requêtes..

CONCEPTS DE PROGRAMMATION: *Type booléen, État de jeu (game state), animation*

CONTRÔLE DU CLAVIER

La commande **getKeyCodeWait()** suspend l'exécution du programme jusqu'à ce qu'une touche du clavier soit actionnée et retourne le code associé à la touche pressée. À l'exception de certaines touches spéciales (Ctrl, Alt, …), chaque touche dispose de son propre code numérique.

On peut déterminer le code associé à chaque touche en utilisant un simple programme de test. Les codes numériques sont **écrits** dans la fenêtre de la console.


```
from gpanel import *
makeGPanel(0, 10, 0, 10)

text(1, 5, "Press any key.")
while True:
    key = getKeyCodeWait()
    print key,
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

```
83 70 72 37 40 38 39
```

MEMENTO

On peut utiliser la fonction **getKeyCodeWait()** pour contrôler les saisies clavier. Celle-ci attend qu'une touche soit pressée et retourne le code associé.

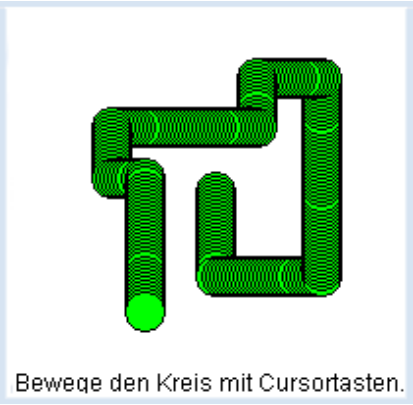
Il faut se rappeler que la fenêtre GPanel doit être active pour pouvoir capturer les événements du clavier. En d'autres termes, la fenêtre du GPanel doit avec le focus. Si la fenêtre perd le focus, il est nécessaire de cliquer quelque part dans son intérieur pour la réactiver. Seule la fenêtre active peut recevoir les événements du clavier.

CONTRÔLER LES FIGURES

Il est possible de déplacer des objets graphiques à l'aide du clavier. Dans l'exemple ci-contre, le programme contrôle le disque vert à l'aide des touches directionnelles du clavier pour le déplacer vers le haut, le bas, la droite ou la gauche. Le programme attend une pression de touche à l'intérieur de la **boucle d'événements**

et gère le code de touche ainsi obtenu dans une structure *if-else* imbriquée dans la boucle.

Du fait que l'affichage du disque est utilisé à plusieurs reprises, il est indiqué d'encapsuler ce code dans une fonction dédiée **drawCircle()** qui peut être réutilisée à de nombreuses reprises, comme l'exige le paradigme de la programmation structurée (*Structured Programming* en anglais).

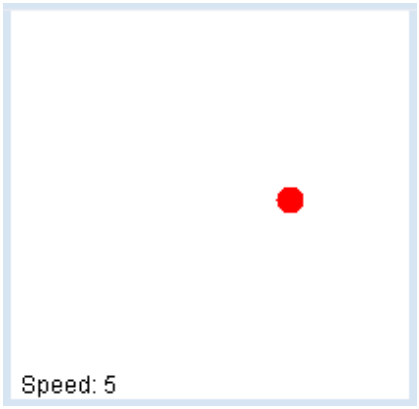



```
from gpanel import *

KEY_LEFT = 37
KEY_RIGHT = 39
KEY_UP = 38
KEY_DOWN = 40

def drawCircle():
    move(x, y)
    setColor("green")
    fillCircle(5)
    setColor("black")
    circle(5)

makeGPanel(0, 100, 0, 100)
text("Move the circle with the arrow keys.")
x = 50
y = 50
step = 2
drawCircle()

while True:
    key = getKeyCodeWait()
    if key == KEY_LEFT:
        x -= step
        drawCircle()
    elif key == KEY_RIGHT:
        x += step
        drawCircle()
    elif key == KEY_UP:
        y += step
        drawCircle()
    elif key == KEY_DOWN:
        y -= step
        drawCircle()
```

MEMENTO

Pour améliorer la lisibilité du programme, on peut introduire des **constants** pour représenter les codes des touches directionnelles. Pour qu'elles soient facilement distinguables des variables, on adopte en *Python* la **convention** de le les écrire en lettres capitales.

INTERROGER LE CLAVIER DE MANIÈRE NON BLOCANTE



Comme vous le savez certainement, le clavier est souvent utilisé pour contrôler le game play dans les jeux vidéo sur ordinateurs. Dans ce cas, il n'est pas possible de recourir à la fonction bloquante *getKeyCodeWait() car elle mettrait le jeu en pause en attendant la pression d'une touche au clavier. Il faut au contraire utiliser une fonction qui fournit le code de la touche actionnée mais qui retourne directement, sans attendre la pression* de la touche.

Dans le cas où une touche a effectivement été pressée, on gère l'événement associé et dans le cas contraire, on continue le jeu normalement.

On veut augmenter ou diminuer la vitesse de la balle à l'aide des touches **F** pour « faster » respectivement **S** pour « slower », mais seulement jusqu'à une vitesse limite. Fixez à nouveau votre attention sur la boucle d'événements où toutes les choses intéressantes se produisent. Cette fonction interroge périodiquement le système pour savoir si une touche a été activée ou non. Si tel est le cas, la fonction **kbhit()** retourne *True* et l'on peut obtenir le code de la touche avec la fonction **getKeyCode()**.

```
from gpanel import *
import time

KEY_S = 83
KEY_F = 70

makeGPanel(0, 600, 0, 600)
title("Key 'f': faster, 's': slower")

enableRepaint(False)
x = 300
y = 300
v = 10
vmax = 50
isAhead = True

while True:
    startTime = time.clock()

    if kbhit():
        key = getKeyCode()
        if key == KEY_F:
            if v < vmax:
              v += 1
        elif key == KEY_S:
            if v > 0:
              v -= 1

    clear()
    setColor("black")
    text(10, 10, "Speed: " + str(v))
    if isAhead:
        x = x + v
    else:
        x = x - v
    move(x, y)
    setColor("red")
    fillCircle(20)
    repaint()
    if x > 600 or x < 0:
        isAhead = not isAhead
```

```
    while (time.clock() - startTime)  < 0.010:
        delay(1)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

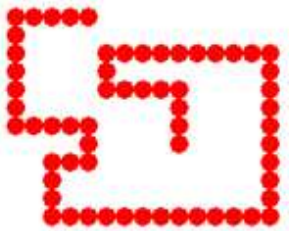
Puisque l'on désire créer une animation, il est nécessaire d'utiliser un **animation timer** pour que le parcours de la boucle soit le plus régulier possible. Le prochain état du jeu est créé dans la boucle et est affiché *a posteriori* dans la fenêtre avec **repaint()**.

kbhit() retourne une valeur de vérité, à savoir une valeur **booléenne**. Si une touche a été enfoncée depuis le dernier appel, la fonction retourne True et elle renvoie False dans tous les autres cas.
Dans le but de déplacer la balle vers la droite (forward), sa coordonnée x doit augmenter de v (la vitesse) lors de chaque passage dans la boucle d'événements. Pour bouger vers la gauche, la coordonnée x doit diminuer. On résume un mouvement avant ou arrière dans un **état de jeu** que l'on stocke dans la variable isAhead.

En Python, on peut ajouter un mot à un second mot à l'aide de l'opérateur +. Ainsi, "Tiger" + "Jython" devient le mot "TigerJython". Par contre, si l'on veut rajouter un nombre à une chaine de caractères, il est nécessaire de convertir le nombre en une chaine de caractères à l'aide de la fonction *str()*.

EXERCICES

1. En utilisant les touches UP, DOWN, LEFT et RIGHT, dessiner un serpent à l'aide de petits disques rouges très serrés.



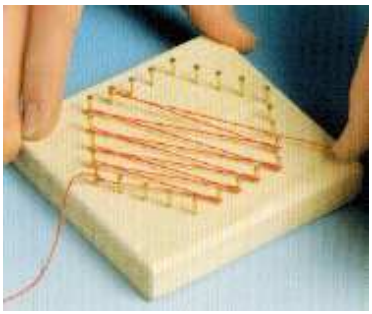
2. Étendre le programme de l'exercice précédent pour qu'il permette de modifier la couleur des prochains disques : la touche *R* sélectionne la couleur rouge « red », *G* la couleur verte « green » et *B* la couleur bleu « blue ».

3. Étendre le programme du ballon se déplaçant de droite à gauche présenté plus haut pour que les touches directionnelles HAUT et BAS déplacent la balle vers le haut et vers le bas.

3.8 ART FILAIRE

INTRODUCTION

Vous avez certainement eu l'occasion de toucher un peu à l'art filaire à l'école enfantine. Rappelez-vous : il s'agissait de planter des clous à intervalles réguliers dans du carton selon un schéma bien défini et de les relier par des ficelles colorées pour faire apparaître de belles figures. Lorsque le nombre de clous était suffisamment grand, des courbes intéressantes se formaient aux endroits où les ficelles se densifiaient. En mathématiques, on parle d'enveloppe de la courbe car les fils sont tangents à la courbe.



De Täubner, Walz: Art filaire

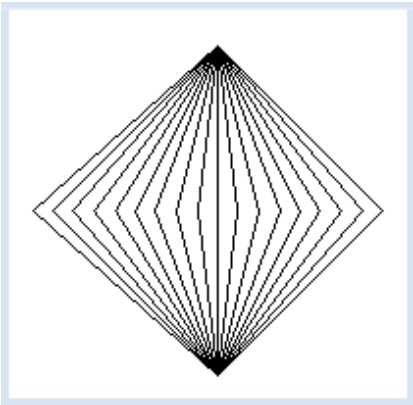
Au lieu de créer ces dessins d'art filaire à la main, on peut aussi confier cette tâche à une machine. Cela impliquerait non seulement que la machine soit capable de comprendre nos instructions, mais encore qu'elle soit capable de les effectuer physiquement, par exemple à l'aide d'un bras robotisé qui puisse tirer des ficelles ou dessiner des traits rectilignes sur un écran. Un tel ensemble d'instructions constitue un **algorithme**. On pourrait imaginer des instructions de réalisation compréhensibles formulées en langage familier. Mais puisqu'il est souhaitable que la machine effectue exactement les mêmes opérations à chaque fois, l'algorithme doit être formulé de manière très précise, de sorte qu'il n'y ait aucune ambiguïté sur les opérations à effectuer à chaque étape. C'est exactement à cette fin que les langages de programmation ont été inventés et c'est la raison pour laquelle vous devez apprendre un langage de programmation. En effet, les langages naturels ne permettent pas d'écrire une suite précise d'opérations de manière non ambiguë.

CONCEPTS DE PROGRAMMATION: *Algorithmes, structures de données, modèle, élégance des programmes, listes, indices.*

DES POINTS EN TANT QUE LISTES

Au lieu de travailler avec des planches, des clous et des ficelles, on va réaliser ces graphiques à l'ordinateur. Pour ce faire, on **modélise** la planche par la fenêtre, les clous par des points et les ficelles par des segments droits.

En transférant l'algorithme dans un langage de programmation, il est indispensable de se rapprocher le plus possible de la réalité. Les clous et les points géométriques représentent des objets tangibles pour l'homme et il devrait en être de même dans le programme.



En géométrie, on peut écrire un point par *P(x,y)* où *x* et *y* sont ses coordonnées cartésiennes. Dans le programme, on peut regrouper ces deux nombres *x* et *y* dans une structure de données appelée **liste**. On écrit pour ce faire, *p = [x, y]*. Le point du plan *P(0, 8)* est donc modélisé par la liste **p = [0, 8]** . On peut accéder aux éléments individuels d'une liste par leur **indice** qui indique leur position dans la liste, en commençant toujours à 0. On écrit l'indice à l'intérieur d'une paire de crochets carrés, à savoir *p[0]* pour la coordonnée *x* et *p[1]* pour la coordonnée *y*. Ce qui est

génial, c'est que l'on peut utiliser cette représentation de points du plan pour toutes les fonctions de GPanel qui nécessitent des coordonnées au lieu de devoir utiliser deux paramètres *x* et *y* indépendants. Le programme ci-dessous modélise le fait de tirer des ficelles par des allers-retours depuis le clou en *A(0,8)* vers le clou en *B(0,-8)* en passant par 19 clous situés sur l'axe des x. Il introduit un délai avec *delay()* pour que le traçage des ficelles soit observable par l'oeil humain.

```
from gpanel import *

DELAY = 100

def step(x):
    p1 = [x, 0]
    draw(p1)
    delay(DELAY)
    draw(pB)
    delay(DELAY)
    p2 = [x + 1, 0]
    draw(p2)
    delay(DELAY)
    draw(pA)
    delay(DELAY)

makeGPanel(-10, 10, -10, 10)
pA = [0, 8]
pB = [0, -8]
move(pA)
for x in range(-9, 9, 2):
    step(x)
```

MEMENTO

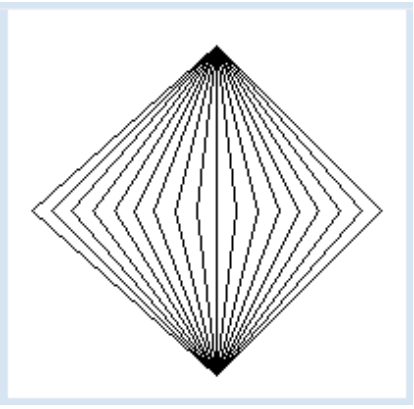
Les données doivent également être structurées de manière appropriée dans l'implémentation d'un algorithme. Nos points sont modélisés par des listes de deux éléments représentant les coordonnées cartésiennes. Le choix de la structure de données affecte de manière significative l'ensemble du programme. Un célèbre professeur d'informatique de l'EPFZ a dit de manière très juste : « Programme = algorithme + structure de données » [**Ref.**]

Les listes peuvent stocker de nombreuses valeurs, appelées **éléments** de la liste, spécifiés dans l'ordre entre crochets carrés. On peut lire les éléments individuels d'une liste grâce à leur indice et leur assigner de nouvelles valeurs. Toutes les commandes graphiques de GPanel fonctionnent également en modélisant les points comme des listes *[x, y]* représentant les coordonnées cartésiennes.

PROGRAMMER EST UN ART

Vous vous êtes probablement dit qu'on pourrait simplifier le précédent programme si on dessinait les lignes sans tenir compte du fait qu'il s'agit de ficelles dans la réalité.

Au lieu de faire des allers-retours entre les points *A* et *B* comme on le ferait à la main avec des ficelles, il suffit de connecter les points *A* et *B* aux points intermédiaires de l'axe des x par des segments droits, ce qui simplifie grandement le programme.

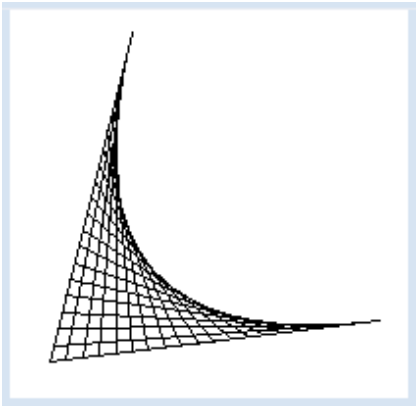


```
from gpanel import *

makeGPanel(-10, 10, -10, 10)
pA = [0, 8]
pB = [0, -8]

for x in range(-9, 10, 1):
    pX = [x, 0]
    line(pA, pX)
    line(pB, pX)
```

MEMENTO

Un algorithme peut être implémenté de différentes manières qui se distinguent de manière significative au niveau de la longueur du code et du temps d'exécution du programme. On peut également parler de programmes plus ou moins élégants. Rappelez-vous seulement pour le moment qu'il n'est pas suffisant pour un programme de produire le résultat correct : **l'élégance et le style** sont également des critères très importants. Considérez la programmation comme un art !

DES ALGORITHMES D'ART FILAIRE ÉLÉGANTS



Lorsque l'on fait de l'art filaire sur l'ordinateur, il est souvent nécessaire de séparer un segment droit en deux segments qui sont dans un rapport bien précis avec la longueur du segment. GPanel met pour ce faire à disposition une fonction **getDividingPoint(pA, pB, r)** à laquelle on passe les deux extrémités *pA* et *pB* du segment à diviser ainsi que le rapport *r* entre les segments *AP* et *AB*. Cette fonction retourne une liste contenant les coordonnées du point délimitant les deux sous-segments

Le programme ci-dessous, particulièrement élégant, modélise un dessin d'art filaire comportant des clous sur les côtés AB et AC..

```
from gpanel import *

makeGPanel(0, 100, 0, 100)

pA = [10, 10]
pB = [90, 20]
pC = [30, 90]

line(pA, pB)
line(pA, pC)

r = 0
while r <= 1:
    pX1 = getDividingPoint(pA, pB, r)
    pX2 = getDividingPoint(pA, pC, 1 - r)
    line(pX1, pX2)
    r += 0.05
    delay(300)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

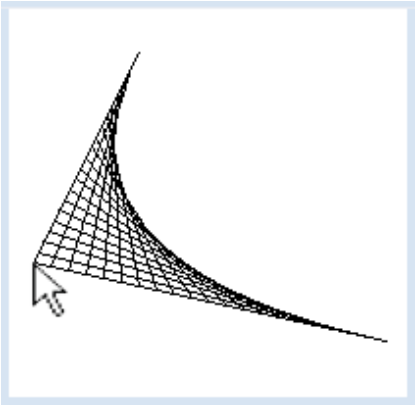
MEMENTO

Les fonctions mises à disposition par des bibliothèques externes, telles que *getDividingPoint(),* peuvent énormément simplifier un programme. Pour effectuer certaines tâches bien délimitées, vous devriez chercher à recourir à des fonctions de bibliothèques existantes que vous avez déjà rencontrées dans votre expérience de programmation, dénichées dans une documentation ou trouvées sur le Web.

Mathématiquement, la courbe émergeant du graphique précédent est une courbe de Bézier quadratique. On peut l'obtenir à l'aide de la fonction *quadraticBezier(pB, pA, pC),* où pB et pC sont les extrémités et pA est le point de contrôle de la courbe. La notion de courbe de Bézier est expliquée en matériel bonus en fin de section.

ART FILAIRE PILOTÉ PAR LA SOURIS

Modeling natural processes with the computer is not just a La modélisation de processus naturels est bien plus qu'un jeu car elle possède des applications très diverses. Il est possible de tester différentes situations facilement et rapidement jusqu'à en trouver une qui soit adaptée à être implémentée en pratique. Le programme est encore plus utile et attractif s'il est possible de changer le modèle en temps réel avec la souris. En *Python*, cela peut se faire très facilement en utilisant des fonctions de rappel. Dans le programme ci-dessous, on peut bouger le sommet A avec la souris.



Afin de créer le graphique, on utilise la fonction **updateGraphics()**qui est appelée par **les gestionnaire d'événements de la souris**. À chaque fois, on efface toute la fenêtre graphique et on la recrée en tenant compte de la nouvelle position du point *A* correspondant à la position du curseur de la souris.

```
from gpanel import *

def updateGraphics():
    clear()
    line(pA, pB)
    line(pA, pC)
    r = 0
    while r <= 1:
        pX1 = getDividingPoint(pA, pB, r)
        pX2 = getDividingPoint(pA, pC, 1 - r)
        line(pX1, pX2)
        r += 0.05

def myCallback(x, y):
    pA[0] = x
    pA[1] = y
    updateGraphics()

makeGPanel(0, 100, 0, 100,
               mousePressed = myCallback,
               mouseDragged = myCallback)

pA = [10, 10]
pB = [90, 20]
pC = [30, 90]
updateGraphics()
```

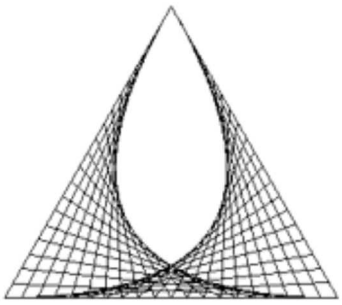
Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

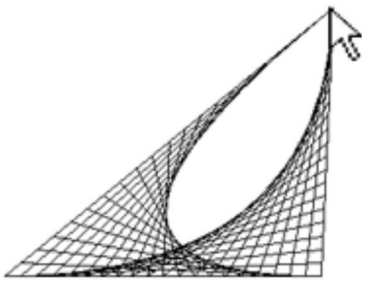
Il est possible sans problème de gérer deux événements différents, en l'occurrence l'événement de clic et l'événement de glissé (drag), en utilisant une seule fonction de rappel en guise de gestionnaire d'événement.

EXERCICES

1. Reproduire le graphique filaire ci-contre



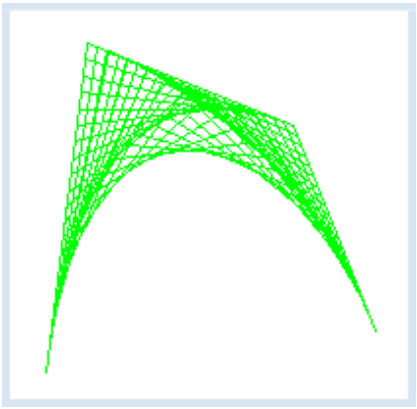
2. Adapter le graphique filaire de l'exercice 1 pour pouvoir glisser le sommet du triangle avec la souris et redessiner le graphique en temps réel avec la nouvelle position du sommet.



MATÉRIEL EN BONUS

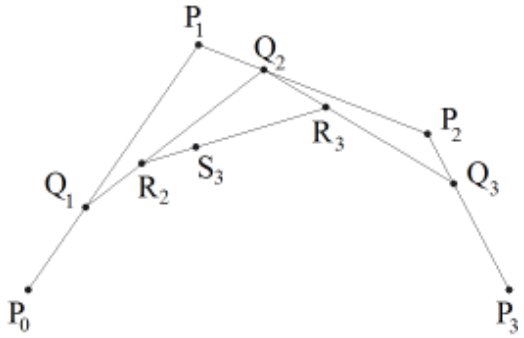
COURBES DE BÉZIER

Ces courbes ont été inventées dans les années soixante du siècle passé par Pierre Bézier qui était alors ingénieur chez Renault. Il cherchait à établir un modèle mathématique pour produire des courbes attirantes dans la conception de produits industriels.



On peut produire des courbes de Bézier par des graphiques filaires en utilisant l'algorithme de Casteljau.The algorithm reads as follows:

Voici les étapes de cet algorithme :



- Spécifier 4 points P0, P1, P2, P3. (P0 et P3 seront les extrémités de la courbe tandis que P1 et P2 serviront de points de contrôle de la courbe).
- Considérer les segments P0P1, P1P2, P2P3
- Partager les segments P0P1, P1P2, P2P3 par les points Q1, Q2, respectivement Q3 dans le même rapport. On a donc l'égalité suivante : P0Q1/P0P1=P1Q2/P1P2=P2Q3/P2P3
- Relier Q1 à Q2 et Q2 à Q3
- Partager les segments Q1Q2 et Q2Q3 par les points R2 et R3 dans le même rapport que pour la troisième étape. On a donc Q1R2/Q1Q2=Q2R3/Q2Q3= P0Q1/P0P1=…
- Relier R2 à R3.

On peut facilement implémenter cet algorithme dans un programme Python en représentant les points par des listes et en appelant plusieurs fois la fonction **getDividingPoint()** au sein de la boucle *while* qui est contrôlée par le rapport *r*.

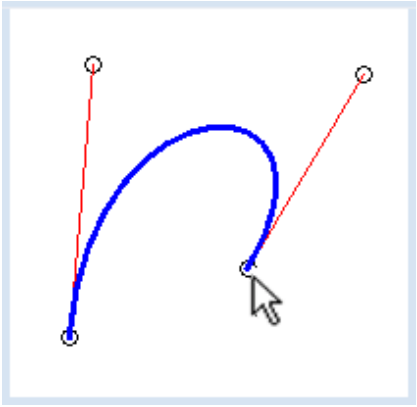
```
from gpanel import *

makeGPanel(0, 100, 0, 100)

pt1 = [10, 10]
pc1 = [20, 90]
pc2 = [70, 70]
pt2 = [90, 20]

setColor("green")

line(pt1, pc1)
line(pt2, pc2)
line(pc1, pc2)

r = 0
while r <= 1:
    q1 = getDividingPoint(pt1, pc1, r)
    q2 = getDividingPoint(pc1, pc2, r)
    q3 = getDividingPoint(pc2, pt2, r)
    line(q1, q2)
    line(q2, q3)
    r2 = getDividingPoint(q1, q2, r)
    r3 = getDividingPoint(q2, q3, r)
    line(r2, r3)
    r += 0.05

setColor("black")
#cubicBezier(pt1, pc1, pc2, pt2)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Une courbe de Bézier cubique est définie par 4 points. Il est possible de dessiner de telles courbes avec la fonction *cubicBezier()* qui utilisera la couleur et l'épaisseur actuellement définies.

DESSIN INTERACTIF DE COURBES



En combinant judicieusement vos connaissances, vous êtes déjà capables d'écrire un programme assez professionnel permettant de créer des courbes de Bézier et de les modifier avec la souris de manière interactive. Le programme ci-dessous est même capable de détecter que le curseur de la souris se trouve à proximité de l'un des points, le colorie et permet de le déplacer en glissant la souris avec le bouton enfoncé.

Le programme doit traiter à de nombreuses reprises les 4 points, raison pour laquelle ils sont stockés dans la liste **points** qu'il est facile de parcourir avec une boucle *for*.

Pour déterminer le point que la souris est en train de manipuler, on utilise la variable **active**: pour stocker la position qu'occupe le point en question au sein de la liste *points*. Si aucun des points n'est manipulé par la souris, *active* prend la valeur -1.

```
from gpanel import *

def updateGraphics():
    # erase all
    clear()

    # draw points
    lineWidth(1)
    for i in range(4):
        move(points[i])
        if active == i:
            setColor("green")
            fillCircle(2)
        setColor("black")
        circle(2)

    # draw tangents
    setColor("red")
    line(points[0], points[1])
    line(points[3], points[2])

    # draw Bezier curve
    setColor("blue")
    lineWidth(3)
    cubicBezier(points[0], points[1], points[2], points[3])

def onMouseDragged(x, y):
    if active == -1:
        return
    points[active][0] = x
    points[active][1] = y
    updateGraphics()

def onMouseReleased(x, y):
    active = -1
    updateGraphics()

def onMouseMoved(x, y):
    global active
    active = near(x, y)
    updateGraphics()

def near(x, y):
    for i in range(4):
```

```
        rsquare = (x - points[i][0]) * (x - points[i][0]) +
                  (y - points[i][1]) * (y - points[i][1])
        if rsquare < 4:
            return i
    return -1

pt1 = [20, 20]
pc1 = [10, 80]
pc2 = [90, 80]
pt2 = [80, 20]
points = [pt1, pc1, pc2, pt2]
active = -1

makeGPanel(0, 100, 0, 100,
    mouseDragged = onMouseDragged,
    mouseReleased = onMouseReleased,
    mouseMoved = onMouseMoved)
updateGraphics()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

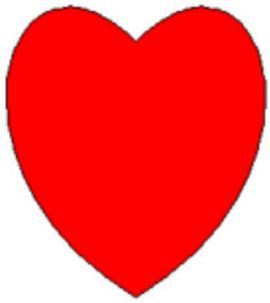
MEMENTO

Il existe des structures de données assez complexes telles que des listes dont les éléments sont eux-mêmes des listes. Dans notre programme, on utilise cela pour stocker une liste de points qui sont eux-mêmes représentés par des listes. On peut alors accéder à la coordonnée *x* du point P1 en utilisant les doubles crochets **points[1][0]**, thus with **double brackets**.

De nos jours, les courbes de Bézier sont utilisées de manière très abondante dans les programmes de dessin vectoriels et de conception assistée par ordinateur (CAO) [**Ref**.]

EXERCICES

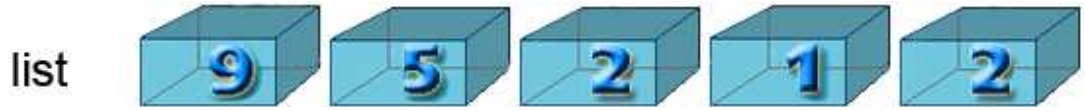
1. Le cœur ci-contre est engendré par deux courbes de Bézier symétriques possédant les mêmes extrémités ainsi que des points de contrôles symétriques. Commencer par dessiner ce cœur sur une feuille de papier en déterminant approximativement la position que doivent prendre les points de contrôle puis réaliser le dessin dans GPanel. Le remplissage est effectué à l'aide de la fonction *fill(point, old_color, new_color),* où *point* représente un point qui se trouve à l'intérieur de la surface du cœur.



3.9 LISTES

INTRODUCTION

Il faut souvent stocker des valeurs qui forment un tout mais dont le nombre n'est pas connu lors de la rédaction du programme. Ceci requière une structure de données capable de stocker plusieurs valeurs et suffisamment flexible pour tenir compte de l'ordre d'insertion des données. La structure Python répondant à ces critères est une séquence de conteneurs simples appelée **liste**. Cette section montre en détails comment travailler avec les listes.



Une liste comportant 5 éléments

Une liste est constituée d'éléments individuels simples arrangés les uns après les autres. Par contraste avec une collection d'éléments non ordonnée, les listes comportent un **premier** et un **dernier** élément. Tous les autres éléments, sauf le dernier, possèdent un **successeur**.

Les listes et les collections de données similaires sont d'une importance capitale en programmation. Les principales opérations possibles sur ces structures de données sont les suivantes :

- Ajouter des éléments (à la fin, au début, quelque part au milieu)
- Lire des éléments
- Modifier des éléments
- Supprimer des éléments
- Itérer sur l'ensemble des éléments
- Trier les éléments
- Rechercher des éléments dans la collection de données

En Python, on peut stocker n'importe quel type de données dans les listes, pas seulement des nombres. Il est même possible que les éléments n'aient pas tous le même type de données. On pourrait par exemple stocker dans une même liste des nombres et des chaines de caractères.

CONCEPTS DE PROGRAMMATION: *Conteneur, liste, prédécesseur, successeur, références*

LISTE DE NOTES

On peut interpréter une liste comme une variable. Elle possède donc un nom (identifiant) et une valeur, à savoir **ses éléments**. On peut créer des listes à l'aide de crochets carrés. L'instruction *li* = [1, 2, 3] va par exemple créer une liste contenant les éléments 1, 2 et 3 dans l'ordre. Une liste peut également ne comporter aucun élément. Une liste vide est symbolisée par *li = []*.

Les carnets de notes où l'on insère les notes correspondant à une matière scolaire donnée sont un exemple typique de liste. Prenons l'exemple des notes de biologie. Au début du semestre, la liste de données est vide, ce qui pourrait s'exprimer en Python par **bioGrades = []**. Ajouter des notes au carnet correspond à l'opération d'ajout d'éléments à la liste. En Python, on utilise la méthode *append()* de sorte que pour une note de 5, on va faire **bioGrades.append(5)**.

On peut visualiser la liste à n'importe quel moment avec une commande *print* en écrivant simplement *print bioGrades*. Si l'on veut calculer la moyenne des notes, il faut parcourir la liste pour additionner toutes les notes. On peut le faire de manière élégante avec une boucle *for* car

for grade in bioGrades:

parcourt la liste *bioGrades* dans l'ordre en copiant chaque élément vers la variable *grade* lors du parcours. Cette variable *grade* est ensuite accessible à l'intérieure du corps de la boucle.

```
bioGrades = []
bioGrades.append(5.5)
print bioGrades
bioGrades.append(5)
print bioGrades
bioGrades.append(5.5)
print bioGrades
bioGrades.append(6)
print bioGrades
sum = 0
for note in bioGrades:
    sum += note
print "Average: " + str(sum / len(bioGrades))
```

MEMENTO

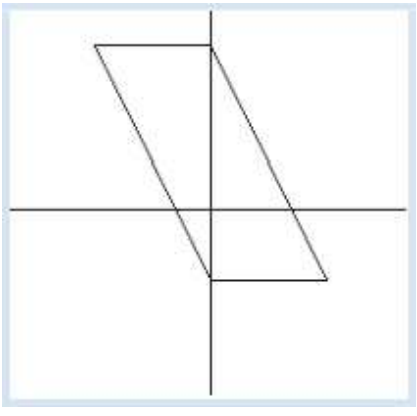
La méthode *append()* permet d'ajouter de nouveaux éléments à la liste.

La fonction intégrée à Python **len(list)** retourne la longueur de la liste *liste*, à savoir le nombre de ses éléments. Notez l'astuce intéressante avec la variable **sum**, qui permet de calculer la somme des notes pour ensuite calculer la moyenne. Au lieu d'utiliser une boucle *for* pour calculer la somme de toutes les notes, on pourrait également utiliser la fonction *sum(bioGrades)* pour la calculer directement.

LISTES COMPORTANT UN NOMBRE FIXE D'ÉLÉMENTS

Souvent, la taille qu'aura une liste est connue d'avance lors du développement du programme et l'on sait également que tous les éléments de la liste auront le même de données. Dans de nombreux langages de programmation, une telle structure de données est appelée **tableau** (*array* en anglais) et on accède aux éléments par le biais de leur indice. En Python, il n'existe pas de tel type de données de longueur fixe et l'on utilise des listes à la place.

Le programme suivant définit un polygone comme une liste de 4 sommets qui sont eux-mêmes définis comme des coordonnées représentées par des listes à deux éléments). Pour pouvoir accéder immédiatement aux éléments à l'aide des indices, on commence par définir le polygone comme une liste d'éléments tous nuls *polygon = [0, 0, 0, 0]*, ce qui peut aussi s'écrire de manière raccourcie par *polygon = [0] * 4*. Ensuite, on remplace les éléments nuls de la liste par les coordonnées des sommets stockés dans les listes à deux éléments. On affiche le polygone à l'aide d'une boucle *for* qui parcourt la liste *polygon*.



```
from gpanel import *

pA = [0, -3]
pB = [5, -3]
pC = [0, 7]
pD = [-5, 7]

makeGPanel(-10, 10, -10, 10)
```

```
line(-10, 0, 10, 0)
line(0, -10, 0, 10)

polygon = [0] * 4  # list with 4 elements, initialized with 0
polygon[0] = pA
polygon[1] = pB
polygon[2] = pC
polygon[3] = pD

for i in range(4):
    k = i + 1
    if k == 4:
        k = 0
    line(polygon[i], polygon[k])
```

MEMENTO

Si l'on connait d'avance la taille de la liste au moment du développement du programme, il est préférable, pour des raisons pratiques et de performance, de créer une liste comprenant directement le bon nombre d'éléments qui seront tous initialisés, généralement avec des zéros. On peut ensuite sans problème se référer à chacun des éléments par son indice.

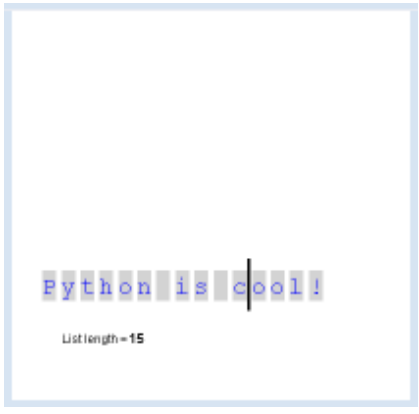
Rappelez-vous que l'indice du premier élément est 0 et celui du dernier élément est *len(liste) – 1*.

INSÉRER ET SUPPRIMER DES ÉLÉMENTS

Le programme suivant montre comment fonctionne un traitement de texte. Les caractères saisis sont insérés dans une liste. Il est clair que la longueur de cette liste n'est pas connue d'avance lors de la rédaction du programme, ce qui fait de la liste une structure de données idéale pour la situation. De plus, on dispose d'un curseur de texte que l'on peut placer n'importe où dans le texte à l'aide de la souris.

Lorsque l'on saisit alors un caractère au clavier, celui-ci est inséré à droite du curseur et la liste est de ce fait allongée. Lorsque l'on utilise la touche « retour arrière », le caractère à gauche du curseur de texte est alors supprimé et la liste est réduite d'un élément..

Pour présenter le texte joliment, on écrit les caractères en texte coloré sur un fond pastel dans GPanel. Pour ce faire, on parcourt la liste sur la base de l'indice *i*.




```
from gpanel import *

BS = 8
SPACE = 32
DEL = 127

def showInfo(key):
    text = "List length = " + str(len(letterList))
    if key != "":
        text += ". Last key code = " + str(ord(key))
    setStatusText(text)

def updateGraphics():
    clear()
```

```
    for i in range(len(letterList)):
        text(i, 2, letterList[i], Font("Courier", Font.PLAIN, 24),
             "blue", "light gray")
    line(cursorPos - 0.2, 1.7, cursorPos - 0.2, 2.7)

def onMousePressed(x, y):
    setCursor(x)
    updateGraphics()

def setCursor(x):
    global cursorPos
    pos = int(x + 0.7)
    if pos <= len(letterList):
       cursorPos = pos

makeGPanel(-1, 30, 0, 12, mousePressed = onMousePressed)

letterList = []
cursorPos = 0
addStatusBar(30)
setStatusText("Enter Text. Backspace to delete. Mouse to set cursor.")
lineWidth(3)

while True:
    delay(10)
    key = getKey()
    if key == "":
        continue
    keyCode = ord(key)
    if keyCode == BS:  # backspace
        if cursorPos > 0:
            cursorPos -= 1
            letterList.pop(cursorPos)
    elif keyCode >= SPACE and keyCode != DEL:
        letterList.insert(cursorPos, key)
        cursorPos += 1
    updateGraphics()
    showInfo(key)
```

MEMENTO

On a déjà vu que les éléments d'une liste sont accessibles par un indice qui commence à 0. Pour cela, on utilise les crochets carrés, à savoir. **letterList[i]**. L'indice doit toujours être compris entre 0 et la longueur de la liste –1. Lorsqu'on utilise une structure **for in range()**, la valeur d'arrêt de la boucle correspond à la longueur de la liste.
Accéder à un élément qui ne se trouve pas dans la liste constitue une erreur comptant parmi les plus fréquentes en programmation. Si l'on ne fait pas très attention lors de l'utilisation des indices, on peut obtenir des programmes qui tantôt fonctionnent et tantôt se plantent.

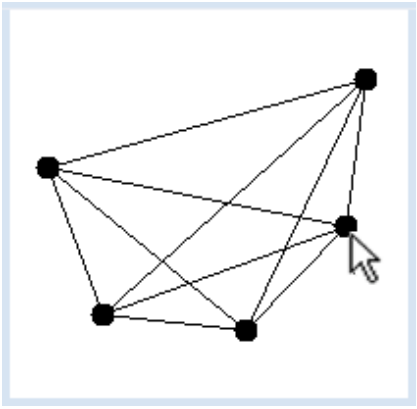
Pour déterminer la touche pressée, on utilise **getKey()**. Cette fonction retourne immédiatement après l'appel et livre soit le dernier caractère saisi au clavier, soit la valeur 65535 (la plus grande valeur entière non signée représentable sur 16 bits) si aucune touche du clavier n'a été pressée.

DÉJÀ UN PROGRAMME PROFESSIONNEL

Vous êtes déjà capables de visualiser un graphe avec les connaissances acquises jusqu'à présent [**plus...**]

Il faut pouvoir créer des petits disques noirs dans la fenêtre graphique à l'aide de clics droits de la souris. Ceux-ci seront considérés comme les sommets d'un graphe qui seront interconnectés par des lignes, appelées arêtes. On aimerait pouvoir déplacer un nœud en cliquant dessus avec le

bouton gauche et en glissant la souris tout en s'assurant que les arêtes connectées à ce graphe soient mises à jour en temps réel. Un clic droit sur un des nœuds du graphe le supprime ainsi que toutes les arêtes correspondantes.

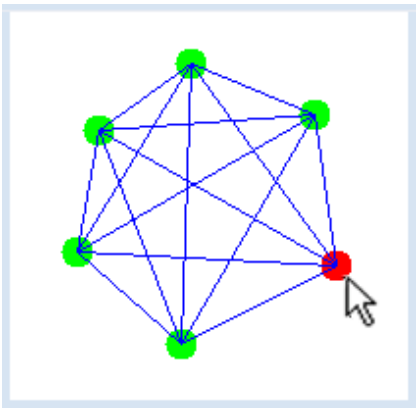


Il est sage, lorsque l'on fait face à une tâche complexe, de s'intéresser en premier lieu à une sous-tâche plus simple mais qui ne remplit pas tous les critères de la spécification du programme. On pourrait par exemple commencer par développer un programme permettant de créer les sommets du graphe à l'aide de clics gauches de la souris en veillant à les connecter à tous les autres nœuds déjà présents, sans qu'il soit pour autant possible de les déplacer.

Il paraît assez évident qu'il faut représenter le graphe par une **liste** dans laquelle on stocke les coordonnées des sommets, toujours sous forme de listes.

Les nœuds sont donc des points *P(x,y)* possédant deux coordonnées que l'on peut représenter à l'aide d'une liste **pt [x, y]**. Le graphe est donc une liste dont les éléments sont des listes de longueur fixée à 2. On va dessiner les arêtes pour connecter les nœuds à l'aide de deux **boucles imbriquées**, mais il faut s'assurer que chaque paire de nœuds n'est reliée que par une seule arête.

```
from gpanel import *

def drawGraph():
    clear()
    for pt in graph:
        move(pt)
        fillCircle(2)

    for i in range(len(graph)):
        for k in range(i, len(graph)):
            line(graph[i], graph[k])

def onMousePressed(x, y):

    pt = [x, y]
    graph.append(pt)
    drawGraph()

graph = []
makeGPanel(0, 100, 0, 100, mousePressed = onMousePressed)
```



Une fois cette fonctionnalité de base implémentée, on peut passer à la fonctionnalité de déplacement par glissé-déposer des nœuds à l'aide du bouton droit de la souris. Lors du glissé-déposé, il est très important de connaitre le nœud qui est en train d'être déplacé. On peut l'identifier par son indice **iNode** dans la liste du graphe. Si aucun nœud n'est en train d'être déplacé, on a **iNode = -1**. La fonction **near(x, y)** utilise le théorème de Pythagore pour calculer la distance entre le point *P(x, y)* du clic de la souris et les sommets du graphe. On utilise cette fonction pour déterminer sur lequel des sommets on est en train de cliquer.

En effet, la souris ne cliquera en général pas exactement sur le milieu des sommets. La fonction

near(x,y) retourne donc l'indice du sommet tel que la distance entre son centre et le clic est inférieur à 10. On remarquera au passage qu'on utilise une instruction *return* en plein milieu de la fonction et non nécessairement à la fin [**plus...**]

Tout le reste du code n'est que petit plaisir de développement que vous auriez aisément pu écrire de manière autonome avec vos connaissances actuelles.

```
from gpanel import *

def drawGraph():
    clear()
    for i in range(len(graph)):
        move(graph[i])
        if i == iNode:
            setColor("red")
        else:
            setColor("green")
        fillCircle(2)

    setColor("blue")
    for i in range(len(graph)):
        for k in range(i, len(graph)):
            line(graph[i], graph[k])

def onMousePressed(x, y):
    global iNode
    if isLeftMouseButton():
        iNode = near(x, y)
    if isRightMouseButton():
        index = near(x, y)
        if index != -1:
            graph.pop(index)
            iNode = -1
        else:
            pt = [x, y]
            graph.append(pt)
    drawGraph()

def onMouseDragged(x, y):
    if isLeftMouseButton():
        if iNode == -1:
            return
        graph[iNode] = [x, y]
        drawGraph()

def onMouseReleased(x, y):
    global iNode
    if isLeftMouseButton():
        iNode = -1
        drawGraph()

def near(x, y):
    for i in range(len(graph)):
        p = graph[i]
        d = (p[0] - x) * (p[0] - x) + (p[1] - y) * (p[1] - y)
        if  d < 10:
            return i
    return -1

graph = []
iNode = -1
makeGPanel(0, 100, 0, 100,
               mousePressed = onMousePressed,
               mouseDragged = onMouseDragged,
               mouseReleased = onMouseReleased)
addStatusBar(20)
setStatusText("Right mouse button to set nodes, left button to drag")
```

MEMENTO

Le programme est complètement **dirigé par les événements**. Le bloc principal ne fait que définir deux variables globales et initialiser la fenêtre de graphiques. Pour chaque action déclenchée par les événements, l'entier de la fenêtre est effacé et redessiné avec les données du graphe mises à jour.

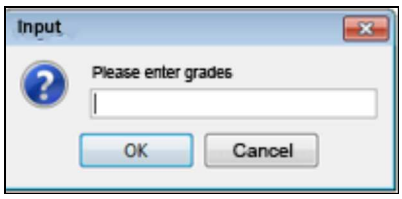
Voici un résumé des opérations les plus importantes sur les listes :

li = [1, 2, 3, 4]	Définit une liste avec les nombres 1, 2, 3, 4
li = [1, "a", [7 , 5]]	Définit une liste avec différents types de données
li[i]	Accède à l'élément d'indice *i*
li[start:end]	Sous-liste d'éléments entre *start* et *end* non compris
li[start:end:step]	Liste d'éléments entre *start* et *end* non compris, par sauts *step*
li[start:]	Sous-liste des éléments à partir de *start*
li[:end]	Sous-liste des éléments depuis le début jusqu'à *end* non compris
li.append(element)	Ajoute un élément en fin de liste
li.insert(i, element)	Insère un élément à la position *i*, repoussant les autres droite
li.extend(li2)	Append tous les éléments de la liste *li2* à la fin de *li*
li.index(element)	Trouve la première occurrence de *element* et retourne son indice
li.pop(i)	Retourne et supprime l'élément d'indice *i*
pop()	Retourne et supprime le dernier élément
li1 + li2	Retourne la concaténation de *li1* et *li2* dans une nouvelle liste
li1 += li2	Remplace *li1* par la concaténation de *li1* et *li2*
li * 4	Nouvelle liste constituée des éléments de *li* répétés 4 fois
[0] * 4	Crée une nouvelle liste de longueur 4 (tous les éléments nuls)
len(li)	Retourne le nombre d'éléments de la liste
del li[i]	Supprimer l'élément d'indice *i*
del li[start:end]	Supprime tous éléments à partir de *start* jusqu'à *end* non compris
del li[:]	Supprimer l'élément d'indice *i*
li.reverse()	Renverse l'ordre de la liste (le dernier devient le premier)
li.sort()	Trie la liste (comparaison avec des méthodes standards)
x in li	Retourne *True* si *x* se trouve dans la liste
x not in li	Retourne *True* si *x* ne se trouve pas dans la liste

La notation [:] avec les crochets carrés et le double-point effectue une opération appelée **slicing** qui coupe une « tranche » dans la liste. *start* et *end* sont des indices de la liste qui fonctionnent exactement comme les paramètres de la fonction *range()*.

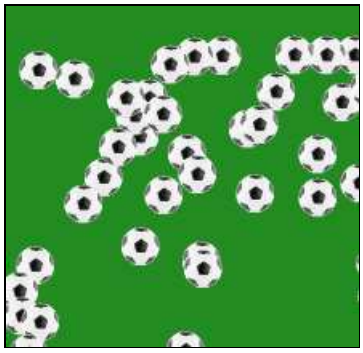
EXERCICES

1. Écrire un programme permettant de saisir un nombre quelconque de notes avec *inputFloat(prompt, False)*. Si l'on clique sur *Annuler*, la moyenne sera calculée et affichée dans la console. Le deuxième paramètre prend la valeur *False* pour éviter que le programme ne se termine si l'on clique sur *Annuler*. Au lieu de cela, la fonction retournera la valeur *None* que l'on peut tester dans une structure *if*.



2. Étendre le programme d'édition de texte vu précédemment avec un slicing de sorte qu'un clic droit supprime le début de la phrase jusqu'au premier espace inclus.

3. Développer un programme faisant apparaître un ballon de football (*football.gif*) à la position de chaque clic de la souris. Tous les ballons sont constamment en mouvement de gauche à droite sur l'écran. Au besoin, replongez-vous dans le chapitre sur les animations qui présente un exemple similaire. On peut optimiser le programme en chargeant une fois pour toutes l'image avec *img = getImage("sprites/football.gif") et passer la variable img à la fonction image().*



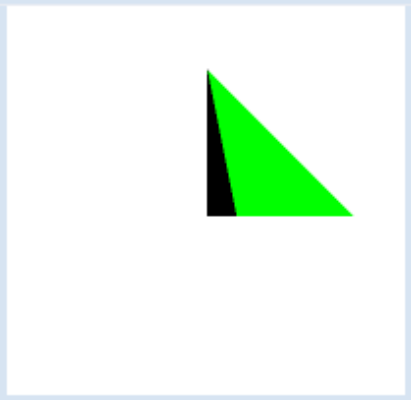
MATÉRIEL SUPPLÉMENTAIRE:

TYPES DE DONNÉES MUTABLES ET IMMUTABLES

En Python, toutes les données sont stockées sous forme d'objets, quel que soit leur type, y compris les types numériques (nombres entiers, flottants, entiers longs et nombres complexes). Comme vous le savez, on peut accéder à un objet par son nom (identifiant). On dit également que le nom **se réfère à** l'objet ou qu'il **est lié** à l'objet (*bound* en anglais). De ce fait, une telle variable est également appelée une **variable référence**.

Un même objet peut être référencé par plus d'un nom. Un nom supplémentaire est appelé un **alias**. L'exemple suivant montre ce concept en action.

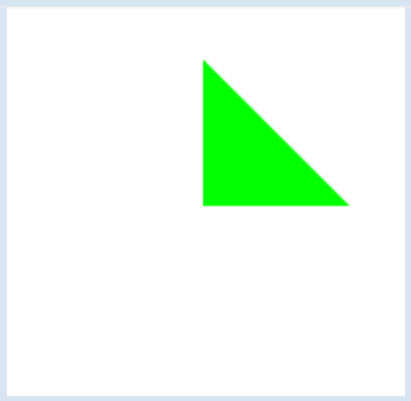
Un triangle est défini par la liste de ses sommets a, b, c. On peut créer un alias avec l'instruction *a_alias* = *a*, de telle sorte que *a* et *a_alias* référencent exactement la même liste de coordonnées. Ainsi, si l'on modifie la liste de coordonnées du sommet dont le nom est *a*, les changements seront également appliqués à la liste de coordonnées référencée par *a_alias* puisque c'est exactement la même..



```
from gpanel import *

makeGPanel(-10, 10, -10, 10)

a = [0, 0]
a_alias = a
b = [0, 5]
c = [5, 0]

fillTriangle(a, b, c)
a[0] = 1
setColor("green")
fillTriangle(a_alias, b, c)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

Du fait que les nombres sont également des objets, on s'attendrait au même comportement si on utilisait des nombres pour représenter les coordonnées des sommets. L'exemple suivant montre cependant qu'il n'en est rien. Si l'on change *xA*, la valeur de *xA_alias* ne change pas.



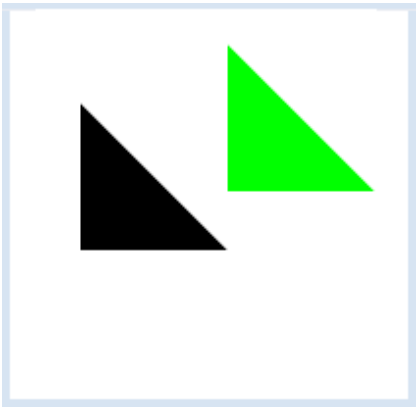
```
from gpanel import *

makeGPanel(-10, 10, -10, 10)

xA = 0
yA = 0
xA_alias = xA
yA_alias = yA
xB = 0
yB = 5
xC = 5
yC = 0

fillTriangle(xA, yA, xB, yB, xC, yC)
xA = 1
setColor("green")
fillTriangle(xA_alias, yA_alias, xB, yB, xC, yC)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

Comment expliquer ce phénomène ? La raison est que les nombres sont des **objets immutables** et que l'instruction *xA + 1* **génère un nouveau nombre**. *xA_alias* vaut donc toujours 0.

La différence entre des types de données immutables et mutables peut également être observée lorsque l'on passe des paramètres aux fonctions. Lorsqu'un objet immutable est passé, il ne peut pas être modifié de l'intérieur de la fonction. Par contre, lorsqu'on passe un objet mutable comme une liste, celui-ci peut être modifié de l'intérieur de la fonction, ce qui constitue un **effet secondaire**, ou **effet de bord**. Lorsque l'on cherche à programmer avec un bon style, on évite le plus possible les effets de bords car ils peuvent causer des comportements non souhaités dont les causes sont souvent très difficiles à identifier.

Dans l'exemple suivant, la fonction *translate()* modifie les listes de coordonnées passées en paramètre pour effectuer une translation.



```
from gpanel import *

def translate(pA, pB, pC):
  pA[0] = pA[0] + 5
  pA[1] = pA[1] + 2
  pB[0] = pB[0] + 5
  pB[1] = pB[1] + 2
  pC[0] = pC[0] + 5
  pC[1] = pC[1] + 2

makeGPanel(-10, 10, -10, 10)

a = [0, 0]
b = [0, 5]
c = [5, 0]
fillTriangle(a, b, c)
translate(a, b, c)
setColor("green")
fillTriangle(a, b, c)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

En Python, tous les types de données sont stockés sous forme d'objets dont certains sont considérés comme immutables. Les types **immutables** en Python sont les suivants: Les types numériques, les chaînes de caractères ainsi que *byte* et *tuple*.
Tous les autres types de données sont **mutables**. Lorsqu'on assigne une nouvelle valeur à une variable immutable, un nouvel objet est créé en mémoire et à son tour référencé par cette variable.

Si des objets mutables sont passés en paramètre à une fonction, la fonction peut les modifier tandis que les objets immutables sont immunisés contre de tels modifications depuis l'intérieur de la fonction.

3.10 HASARD ET NOMBRES ALÉATOIRES

INTRODUCTION

La chance joue un rôle important dans notre vie de tous les jours. On parle de « chance » ou de « hasard » pour désigner les événements qui ne sont pas prédictibles. Si quelqu'un qui ne vous connaît absolument pas vous demande de choisir parmi les couleurs rouge, vert ou bleu, il sera incapable de prévoir à l'avance quelle couleur vous allez choisir, ce qui confère à ce choix un caractère aléatoire. Le hasard joue un rôle très important dans les jeux : lorsque l'on jette un dé, le nombre de points que l'on obtient entre 1 et 6 est complètement aléatoire si le dé n'est pas truqué.
Bien que le monde soit régi par le hasard, il n'est cependant pas chaotique puisque même le hasard présente certaines régularités qui permettent un certain degré de prédictibilité. Il faut toutefois bien souligner qu'il s'agit de prédications « en moyenne », à savoir si certaines situations se reproduisent à de nombreuses reprises. Dans le but d'explorer les lois du hasard, il faut mettre en place des **expériences stochastiques** (aléatoires) dont on définit précisément les conditions initiales mais dont le processus est ensuite dirigé par les nombres aléatoires.

L'ordinateur se prête à merveille à la réalisation d'expériences stochastiques car il permet de faire un nombre important d'expériences. Pour cela, l'ordinateur va devoir générer une série de nombres aléatoires qui sont le plus indépendants possible les uns des autres. On utilise le plus souvent des nombres aléatoires entiers pris dans un certain intervalle, par exemple entre 1 et 6, ou des nombres réels compris entre 0 et 1. Un algorithme capable de générer une suite de nombres aléatoires est appelé **générateur de nombres aléatoires**. Il est très important que tous les nombres générés apparaissent avec la même fréquence à la manière d'un dé non truqué. On dit de tels nombres qu'ils forment une **distribution uniforme** ou qu'ils sont **uniformément distribués**.

CONCEPTS DE PROGRAMMATION: *Nombres aléatoires, expériences stochastiques, fréquence d'apparition, probabilités*

DESSINS ALÉATOIRES

On étale 20 ellipses colorées de taille, de position et de couleur aléatoires sur un canevas. Le lecteur jugera par lui-même si ce processus relève de la peinture ou de l'œuvre d'art. Quoi qu'il en soit, les figures qui résultent de ce processus complètement aléatoire sont sympathiques. Pour déterminer la position et la taille des ellipses, on peut utiliser la fonction **random()** du **random module**, Celle-ci va livrer un nouveau nombre aléatoire compris entre 0 et 1 à chaque appel. Pour obtenir les couleurs aléatoires, il faut **trois nombres aléatoires compris entre 0 et 255** pour déterminer les niveaux d'intensité des trois couleurs fondamentales rouge, vert et bleu qui entreront dans la composition de la couleur.

```
from gpanel import *
import random
```

```
def randomColor():
    r = random.randint(0, 255)
    g = random.randint(0, 255)
    b = random.randint(0, 255)
    return makeColor(r, g, b)

makeGPanel()
bgColor(randomColor())

for i in range(20):
    setColor(randomColor())
    move(random.random(), random.random())
    a = random.random() / 2
    b = random.random() / 2
    fillEllipse(a, b)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

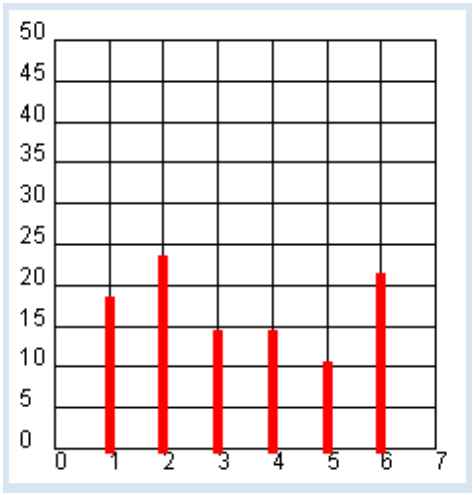
La fonction **random.random()** retourne des nombres flottants uniformément distribués entre 0 (inclus) et 1 (exclus). Il est nécessaire d'**importer** le module *random* pour pouvoir y accéder. Comme nous le verrons en détails dans la section « Traitement d'images » de ce chapitre, les couleurs sont définies par les trois composantes fondamentales rouge, vert et bleu (RVB = RGB = Red, Green, Blue) qui sont combinées par synthèse additive. Les valeurs de chaque composante de couleur sont comprises entre 0 et 255.

La fonction **randint(start, end)** renvoie un nombre aléatoire entier compris entre *start* et *end* (Les deux bornes sont incluses). La fonction *makeColor()* retourne un objet-couleur résultant de la composition des composantes rouge, vert et bleu.

FRÉQUENCE D'APPARITION AU JET D'UN DÉ

Une expérience aléatoire très commune consiste à jeter un dé à six faces 100 fois pour déterminer la fréquence d'apparition des nombres 1, 2, …, 6.

On peut utiliser l'ordinateur pour accélérer très nettement cette expérience. On remplacera simplement le dé par des **nombres aléatoires entre 1 et 6**. On peut représenter graphiquement la distribution des fréquences d'apparition dans GPanel..




```
from gpanel import *
import random

NB_ROLLS = 100

makeGPanel(-1, 8, -0.1 * NB_ROLLS / 2, 1.1 * NB_ROLLS / 2)
title("# Rolls: " + str(NB_ROLLS))
drawGrid(0, 7, 0, NB_ROLLS // 2, 7, 10)
setColor("red")

histo = [0, 0, 0, 0, 0, 0, 0]
# hist = [0] * 7  # short form
```

```
for i in range(NB_ROLLS):
    pip = random.randint(1, 6)
    histo[pip] += 1

lineWidth(5)
for n in range(1, 7):
    line(n, 0, n, histo[n])
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

On utilise la liste **histo** pour stocker les fréquences d'apparition des différents événements aléatoires (résultats des dés) en incrémentant l'élément de la liste dont l'indice correspond au résultat obtenu. Pour s'épargner d'inutiles calculs sur les indices, on utilisera une liste de 7 éléments dont le premier, d'indice 0, ne sera jamais utilisé.

Quelques expériences de ce type révèlent rapidement que les fréquences d'apparition deviennent de plus en plus uniformes à mesure que le nombre de jets **NB_ROLLS**. Saugmente, de sorte qu'elles tendent toutes de plus en plus vers 1/6. Mathématiquement, on exprime ce fait de la manière suivante : **les issues 1, 2, 3, 4, 5, 6 de l'expérience aléatoire sont équiprobables et leur probabilité d'apparition vaut 1/6**.

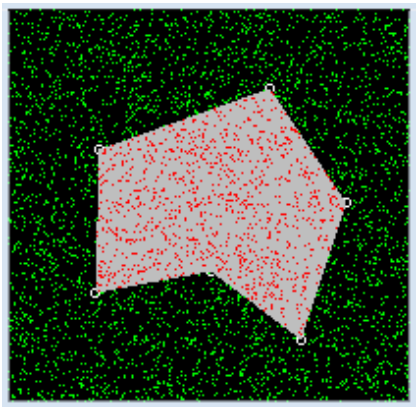
Pour dessiner la grille du repère, il faut faire appel à la fonction *drawGrid(xmin, xmax, ymin, ymax, xticks, yticks)* comprenant 6 paramètres dont les deux derniers définissent le nombre de subdivisions. Si *xmax* ou *ymax* est un nombre flottant, les axes seront libellés avec des nombres flottants. Dans le cas contraire, les axes seront libellés avec des nombres entiers.

SIMULATIONS MONTE CARLO

La principauté de Monaco est très célèbre pour ses casinos. Ceux-ci n'ont pas seulement été une attraction pour les célébrités durant les 150 dernières années mais aussi pour les mathématiciens qui tentent d'analyser les jeux de hasard pour échafauder des stratégies de jeu gagnantes. L'ordinateur est bien plus adapté pour tester ces stratégies et plus conciliant que le véritable jeu puisque l'on peut réaliser l'expérience aléatoire sans y laisser des fortunes.

Dans le « jeu » suivant, on place des points sur une surface carrée contenant un polygone. On pourrait comparer chaque point à l'impact d'une goutte de pluie. Lorsqu'il pleut, les gouttes sont en principe **distribuées de manière uniforme** : chaque unité de surface comptera environ le même nombre de gouttes. Pour revenir à notre polygone, on laisse tomber un certain nombre de gouttes sur notre carré et on compte le nombre de gouttes tombées à l'intérieur du polygone. Il est évident que ce nombre sera d'autant plus grand que la surface du polygone sera importante et qu'en moyenne (pour autant qu'il y ait un nombre très élevé de gouttes), ce nombre sera même proportionnel à la surface. Ainsi, si l'aire du polygone vaut ¼ de celle du carré qui l'entoure, ¼ des gouttes tombées dans le carré auront atteint le polygone. Une fois ceci établi, on peut renverser la vapeur pour déterminer le rapport entre la surface du polygone et celle du carré, facilement calculable, en calculant simplement le rapport entre le nombre de gouttes tombées dans le polygone et celles tombées sur l'ensemble du carré.

Le programme suivant est conçu pour être moderne et intuitif. Un clic gauche de la souris permet de définir les sommets du polygone. On peut ensuite désigner la surface dont on aimerait calculer l'aire avec un clic droit, ce qui va dessiner le polygone et faire « pleuvoir » des points aléatoires sur le canevas.
Le rapport entre le nombre de points tombés dans le polygone et ceux tombés sur l'ensemble du canevas sera affiché dans la barre de titre.



```
from gpanel import *
import random

NB_DROPS = 10000

def onMousePressed(x, y):
    if isLeftMouseButton():
        pt = [x, y]
        move(pt)
        circle(0.5)
        corners.append(pt)
    if isRightMouseButton():
        wakeUp()

def go():
    global nbHit
    setColor("gray")
    fillPolygon(corners)
    title("Working. Please wait...")
    for i in range(NB_DROPS):
        pt = [100 * random.random(), 100 * random.random()]
        color = getPixelColorStr(pt)
        if color == "black":
            setColor("green")
            point(pt)
        if color == "gray" or color == "red":
            nbHit += 1
            setColor("red")
            point(pt)
    title("All done. #hits: " + str(nbHit) + " of " + str(NB_DROPS))

makeGPanel(0, 100, 0, 100, mousePressed = onMousePressed)
title("Select corners with left button. Start dropping with right button")
bgColor("black")
setColor("white")
corners = []
nbHit = 0
putSleep()
go()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Lors du **clic gauche de la souris**, le programme sauve le sommet ainsi créé dans la liste *corners* et l'entoure d'un petit cercle servant de marqueur.

La simulation de la pluie est effectuée dans la fonction **go()**. Elle débute lors d'un clic droit et dure un certain laps de temps. Les impacts des gouttes de pluie sont marqués par des points colorés de couleurs différentes suivant qu'ils tombent à l'intérieur ou à l'extérieur du polygone.

Si l'on appelait directement la fonction *go()* depuis la fonction *pressCallback()*, comme il peut paraître évident de le faire, on ne pourrait voir aucun point avant que la simulation ne se termine. Cela vient du fait que le système empêche le rafraîchissement du graphique depuis un gestionnaire d'événement de la souris pour des raisons qui lui sont propres. Pour visualiser une action de longue durée, il faut donc placer le code ailleurs que dans la fonction de rappel, en général dans le programme principal. L'exécution du programme principal est mise en pause avec **putSleep().** Le clic droit réveille le programme principal avec **wakeUp()**, ce qui va lancer la simulation avec l'appel à *go()*.

De manière générale, pour éviter des surprises désagréables, il faut toujours respecter la règle suivante:

Les fonctions de rappel (gestionnaires d'événements) doivent impérativement retourner rapidement à l'appelant. De ce fait, aucune action de longue durée ne devrait prendre place à l'intérieur d'une fonction de rappel, sans quoi l'exécution du programme s'en trouverait bloquée.

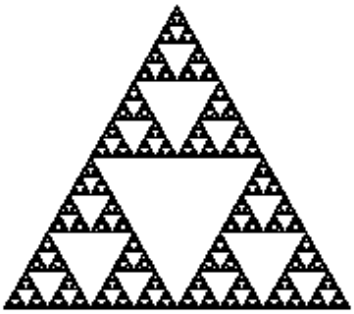
Pour déterminer si une goutte est tombée à l'intérieur du polygone ou non, il suffit de consulter la couleur du pixel correspondant avec **getPixelColorStr()** puisque les pixels du polygone sont gris et les autres noirs. Il faut tenir compte également du fait qu'une goutte peut tomber sur un pixel rouge si celui-ci a déjà été précédemment atteint par une goutte. De ce fait, si le pixel atteint est gris ou rouge, on incrémente **nbHit by 1** et on colorie le pixel en rouge. Vous pouvez tester cette procédure en générant quelques polygones simples (par exemple des triangles ou des rectangles) et en les mesurant à l'écran avec une règle. Vous constaterez qu'il faut un nombre important de point pour que l'aire estimée avec Monte Carlo corresponde bien à l'aire réelle [**plus...**].

JEU DU CHAOS

Il peut paraître surprenant de pouvoir créer des motifs réguliers par des expériences aléatoires. Cela vient de la compensation des fluctuations statistiques par un grand nombre d'expériences. Michael Barnsley inventa en 1988 l'algorithme suivant, basé sur la théorie du chaos, qui s'appuie sur une sélection aléatoire des sommets d'un triangle :

1. Construire un triangle équilatéral de sommets A, B, C
2. Choisir un point P à l'intérieur du triangle ABC
3. Sélectionner aléatoirement un des sommets
4. Placer le point P' qui partager en deux parts égales le segment qui relie P et le sommet choisi
5. Dessiner le point P
6. Répéter les étapes 2, 3, 4, 5

Une telle formulation tient la route dans le langage courant mais elle ne peut pas être directement traduite en un programme informatique puisque l'étape 6 demande de retourner à l'étape 3. En effet, les langages de programmation modernes ne possèdent pas d'instruction permettant de sauter à un autre endroit du programme comme ce fut jadis le cas avec les instructions *goto*. Les sauts sont de nos jours implémentés avec des boucles [**plus...**].



```
from gpanel import *
import random
```

```
MAXITERATIONS = 100000
makeGPanel(0, 10, 0, 10)

pA = [1, 1]
pB = [9, 1]
pC = [5, 8]

triangle(pA, pB, pC)
corners = [pA, pB, pC]
pt = [2, 2]

title("Working...")
for iter in range(MAXITERATIONS):
    i = random.randint(0, 2)
    pRand = corners[i]
    pt = getDividingPoint(pRand, pt, 0.5)
    point(pt)
title("Working...Done")
```

MEMENTO

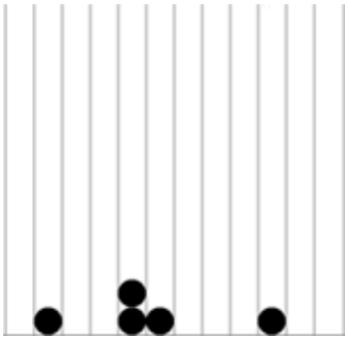
Si l'on a besoin de choisir aléatoirement un objet parmi plusieurs, on les place tous dans une **list** and pick an object out of it at a random index.

Il est assez étonnant que l'on puisse engendrer une figure (appelée **triangle de Sierpinski**) présentant une grande régularité en choisissant des points aléatoires.

EXERCICES

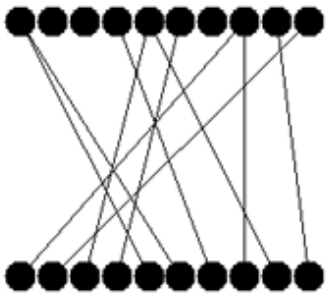
1. Cinq enfants se rencontrent à la place de jeux et se demandent les uns aux autres leur mois de naissance. Il est assez surprenant que la probabilité qu'aux moins deux d'entre eux aient le même mois d'anniversaire soit assez élevée.

   Créer une simulation effectuant 100 expériences aléatoires pour déterminer expérimentalement cette probabilité. Illustrer cela en montrant pour chaque expérience 12 rectangles représentant les mois de l'année. Lorsqu'un enfant est né dans un certain mois, on rajoutera un disque dans le rectangle correspondant. Le résultat de la simulation sera écrit dans la barre de titre de la fenêtre.



2. Vingt enfants jouent à un jeu de ballon consistant, pour la première équipe de dix enfants, à jeter chacun un ballon sur un des dix enfants de l'équipe adverse, au hasard et simultanément. On considère qu'il n'y a pas d'interaction entre les balles jetées. Les enfants qui sont touchés sont éliminés. En moyenne, combien y a-t-il d'enfants de l'équipe adverse qui demeurent intouchés ?

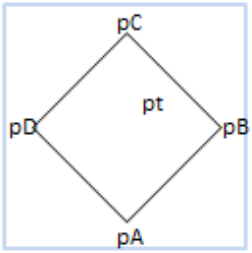
   Créer une simulation comportant 100 expériences aléatoires pour déterminer ce nombre expérimentalement. Illustrer ensuite chacune des expériences en représentant les membres des équipes par des disques pleins et la trajectoire des ballons par des segments droits. Le résultat de l'ensemble des expériences sera écrit dans la barre de titre de la fenêtre.



3. On peut déterminer l'aire de n'importe quelle surface à l'aide d'une simulation Monte Carlo. Modifier le programme développé dans cette section pour dessiner une surface arbitraire en maintenant le bouton gauche de la souris enfoncé. Un clic droit à l'intérieur de cette surface la coloriera dans une autre couleur et lancera la simulation.

4. Simuler le jeux du chaos en utilisant un carré de sommets pA(0, -10), pB(10, 0), pC(0, 10), pD(-10, 0) et en prenant aléatoirement n'importe quel point *pt* à l'intérieur du carré.

   Diviser le segment reliant un des sommets aléatoirement choisi et le point aléatoire *pt* avec un facteur de division de 0.45. Autrement dit, colorier le point *P'* tel que *ptP'=0.45* * *ptP*, *P* étant le sommet aléatoirement choisi.



3.11 TRAITEMENT D'IMAGES

INTRODUCTION

Les humains interprètent une image comme une surface plane généralement rectangulaire comportant des formes colorées. En informatique et dans l'industrie de l'impression, une image est plutôt vue comme une grille de points colorés appelés **pixels**. Le nombre de pixels par unité de surface est appelé résolution de l'image et est souvent indiquée en ppp (points par pouces). En anglais, on parle de dpi (*dots per inch*).

Pour pouvoir stocker et traiter une image dans un ordinateur, il faut définir les couleurs par des valeurs numériques. Il y a plusieurs possibilités de le faire qui sont appelés **modèles de couleurs** ou **systèmes colorimétriques** . Un des modèles les plus populaires est RGB qui représente l'intensité de chacune des trois couleurs fondamentales rouge, vert et bleu par une échelle comprise entre 0 (pas de couleur) et 255 (intensité maximale). [**plus...**]. Le modèle ARGB inclut encore un autre nombre compris entre 0 et 255 qui indique la transparence (valeur alpha) de la couleur [**plus...**].

En bref : une image est représentée dans un ordinateur comme un tableau bidimensionnel de nombres représentant chacun la couleur d'un pixel. Cette représentation d'une image est appelée un **bitmap**.

CONCEPTS DE PROGRAMMATION: *Numérisation d'images, résolution, modèles de couleur, bitmap, format d'images.*

MÉLANGES DE COULEURS DANS LE MODÈLE

TigerJython met à disposition des objets de type *GBitmap* pour simplifier la manipulation d'images bitmap. L'instruction **bm = GBitmap(width, height)** génère un bitmap comportant le nombre indiqué de pixels en hauteur et en largeur. Il est ensuite possible de modifier la couleur de chacun des pixels de manière individuelle avec la méthode **setPixelColor(x, y, color)** et de lire leur couleur avec la méthode *getPixelColor(x, y)*. Finalement, la méthode *image()* permet de dessiner le bitmap dans un canevas GPanel.

Le programme ci-dessous utilise un objet *GBitmap* pour dessiner les fameux disques de la synthèse additive des trois couleurs fondamentales en parcourant le bitmap avec une boucle imbriquée.

```
from gpanel import *

xRed = 200
yRed = 200
xGreen = 300
yGreen = 200
xBlue = 250
yBlue = 300

makeGPanel(Size(501, 501))
window(0, 501, 501, 0)     # y axis downwards
```

```
bm = GBitmap(500, 500)
for x in range(500):
  for y in range(500):
      red = green = blue = 0
      if (x - xRed) * (x - xRed) + (y - yRed) * (y - yRed) < 16000:
         red = 255
      if (x - xGreen) * (x - xGreen) + (y - yGreen) * (y - yGreen) < 16000:
         green = 255
      if (x - xBlue) * (x - xBlue) + (y - yBlue) * (y - yBlue) < 16000:
         blue = 255
      bm.setPixelColor(x, y, makeColor(red, green, blue))

image(bm, 0, 500)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Les couleurs sont définies par leurs composantes de rouge, vert et bleu. La fonction *makeColor(red, green, blue)* permet de combiner ces composantes pour former un objet couleur.

Les images utilisent typiquement un système de coordonnées où l'origine se trouve au coin supérieur gauche, l'axe *y* pointant vers le bas [**plus...**].

CREER UNE IMAGE EN NIVEAUX DE GRIS

Vous vous êtes peut-être déjà demandé comment un logiciel de traitement d'images tel que Photoshop fonctionne. On va voir ensemble quelques notions de base du traitement d'images qui permettront de comprendre les principes élémentaires. Le programme suivant va transformer une image colorée en nuances de gris. On utilise pour ce faire la moyenne des composantes de rouge, vert et bleu pour déterminer la valeur de la nuance de gris.

```
from gpanel import *

size = 300

makeGPanel(Size(2 * size, size))
window(0, 2 * size, size, 0)     # y axis downwards
img = getImage("sprites/colorfrog.png")
w = img.getWidth()
h = img.getHeight()
image(img, 0, size)
for x in range(w):
    for y in range(h):
        color = img.getPixelColor(x, y)
        red = color.getRed()
        green = color.getGreen()
```

```
        blue = color.getBlue()
        intensity = (red + green + blue) // 3
        gray = makeColor(intensity, intensity, intensity)
        img.setPixelColor(x, y, gray)
image(img, size, size)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

On peut déterminer les composantes d'un objet couleur en utilisant les méthodes **getRed()**, *getGreen(), getBlue().*

Le fond doit être blanc et non transparent. Pour permettre la transparence, on peut déterminer la valeur de transparence avec *alpha = getAlpha()* et utiliser ensuite cette valeur comme paramètre supplémentaire de *makeColor(red, green, blue, alpha)*.

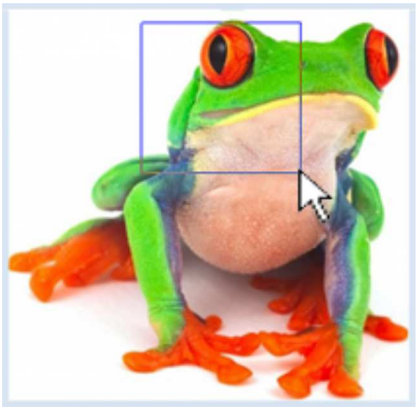
RÉUTILISABILITÉ

Dans la plupart des programmes de traitement d'images, l'utilisateur doit être en mesure de sélectionner une portion rectangulaire de l'image. Pour cela, on peut dessiner un rectangle temporaire « élastique » en glissant la souris. Lorsque le bouton de la souris est relâché, la zone est définitivement choisie. Un programmeur avisé commencera par résoudre ce problème récurrent avant de s'attaquer au développement du logiciel dans son ensemble puisque cette fonctionnalité sera réutilisée pour implémenter de nombreuses fonctions et applications différentes. La réutilisabilité est un des critères de qualité majeurs dans tous les domaines du développement logiciel.

Comme nous l'avons déjà vu, le dessin de lignes « élastiques » peut être considéré comme une animation. Dans le cas qui nous intéresse, cela impliquerait cependant de redessiner l'image dans sa totalité à chaque changement du rectangle de sélection. Une astuce très performante pour éviter cela consiste à utiliser **le mode de dessin XOR**. Ce mode a la particularité de combiner la figure en cours de dessin avec les pixels déjà présents dans le canevas. Deux dessins successifs de la même figure en mode XOR la font disparaître sans pour autant affecter les pixels sous-jacents. Le petit bémol de ce procédé réside dans le changement de couleur incontrôlable pendant le dessin, ce qui ne se remarque toutefois presque pas pour une surface aussi faible que le bord d'un rectangle de sélection.

Dans le code ci-dessous, la fonction **doIt()** est appelée seulement après que le rectangle de sélection a été déterminé. Elle écrit les coordonnées du coin supérieur gauche du rectangle de sélection, *ulx* et *uly* (*upper left x, y)* et de son coin inférieur droit *lrx*, *lry* (*lower right x, y)*. Cette fonction *doIt()* accueillera ultérieurement le code du traitement à appliquer sur la sélection.

Vous devriez être en mesure de comprendre sans problème ce code avec vos connaissances préalables à propos des événements.



```
from gpanel import *

size = 300

def onMousePressed(e):
    global x1, y1
```

```
    global x2, y2
    setColor("blue")
    setXORMode(Color.white) # set XOR paint mode
    x1 = x2 = e.getX()
    y1 = y2 = e.getY()

def onMouseDragged(e):
    global x2, y2
    rectangle(x1, y1, x2, y2) # erase old
    x2 = e.getX()
    y2 = e.getY()
    rectangle(x1, y1, x2, y2) # draw new

def onMouseReleased(e):
    rectangle(x1, y1, x2, y2) # erase old
    setPaintMode() # establish normal paint mode
    ulx = min(x1, x2)
    lrx = max(x1, x2)
    uly = min(y1, y2)
    lry = max(y1, y2)
    doIt(ulx, uly, lrx, lry)

def doIt(ulx, uly, lrx, lry):
    print "ulx = ", ulx, "uly = ", uly
    print "lrx = ", lrx, "lry = ", lry

x1 = y1 = 0
x2 = y2 = 0

makeGPanel(Size(size, size),
    mousePressed = onMousePressed,
    mouseDragged = onMouseDragged,
    mouseReleased = onMouseReleased)
window(0, size, size, 0)    # y axis downwards

img = getImage("sprites/colorfrog.png")
image(img, 0, size)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

On peut récupérer le bitmap d'une image enregistrée sur l'ordinateur avec la fonction **getImage()** qui demande soit le chemin d'accès complet au fichier ou uniquement un chemin relatif par rapport au dossier contenant le programme Python. Pour charger les images se trouvant dans l'archive JAR de la distribution TigerJython, il faut utiliser le dossier spécial *sprites*.

Dans le gestionnaire de l'événement de clic de la souris, on bascule le système de dessin en **XOR mode** pour permettre au gestionnaire de glissé de la souris *onMouseDragged* de supprimer le vieux rectangle en le redessinant une deuxième fois pour ensuite le redessiner avec les nouvelles coordonnées de la souris. Les coordonnées *x1, y1, x2, y2* doivent être stockées dans des variables globales pour permettre la communication entre les gestionnaires d'événements. D'autre part, si l'on redessinait le rectangle de sélection lorsque le bouton de la souris est relâché avant de repasser au mode de dessin normal, il disparaîtrait. Il faut donc commencer par repasser au mode de dessin normal pour éviter qu'il ne disparaisse.

Le programme est assez flexible pour retourner les coordonnées *ulx,uly* et *lrx, lry* quelle que soit la manière de dessiner le rectangle de sélection, même si l'on commence par exemple par le coin supérieur droit. Ainsi, on aura toujours *ulx < lrx et uly < lry*. Notez que le programme n'effectue aucune conversion entre les coordonnées de la souris (mouse coordinates) et les coordonnées fenêtre (window coordinates) puisque les deux systèmes coïncident dans le mesure où l'on choisit une taille identique pour la fenêtre avec *size()* et le système de

coordonnées avec *window()*.Même lorsque la souris est déplacée à l'extérieur de la fenêtre, le programme continue de recevoir des événements de type « glissé de la souris ». Il faut être très prudent à l'utilisation de ces coordonnées extérieures à la fenêtre, sans quoi le programme pourrait planter de manière inattendue.

SUPPRESSION DES YEUX ROUGES

Le traitement d'image joue un rôle très important dans la retouche de photos prises avec un appareil numérique. Internet regorge de tels programmes mais pourrez bientôt vous en passer et, armés de Python, d'une bonne dose d'imagination et de persévérance, développer des programmes qui correspondent exactement à vos besoins. Le programme ci-dessous supprime l'effet des yeux rouges sur une photo numérique survenant à cause de la réflexion du flash sur le fond d'œil (fundus oculi). On utilisera en l'occurrence l'image d'une grenouille qui est intéressante puisqu'elle comporte d'autres zones rouges que les yeux.

```
from gpanel import *

size = 300

def onMousePressed(e):
    global x1, y1
    global x2, y2
    setColor("blue")
    setXORMode("white")
    x1 = x2 = e.getX()
    y1 = y2 = e.getY()

def onMouseDragged(e):
    global x2, y2
    rectangle(x1, y1, x2, y2) # erase old
    x2 = e.getX()
    y2 = e.getY()
    rectangle(x1, y1, x2, y2) # draw new

def onMouseReleased(e):
    rectangle(x1, y1, x2, y2) # erase old
    setPaintMode()
    ulx = min(x1, x2)
    lrx = max(x1, x2)
    uly = min(y1, y2)
    lry = max(y1, y2)

    doIt(ulx, uly, lrx, lry)

def doIt(ulx, uly, lrx, lry):
    for x in range(ulx, lrx):
        for y in range(uly, lry):
            col = img.getPixelColor(x, y)
            red = col.getRed()
            green = col.getGreen()
            blue = col. getBlue()
            col1 = makeColor(3 * red // 4, green, blue)
            img.setPixelColor(x, y, col1)
    image(img, 0, size)

x1 = y1 = 0
x2 = y2 = 0
```

```
makeGPanel(Size(size, size),
    mousePressed = onMousePressed,
    mouseDragged = onMouseDragged,
    mouseReleased = onMouseReleased)
window(0, size, size, 0)     # y axis downwards

img = getImage("sprites/colorfrog.png")
image(img, 0, size)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

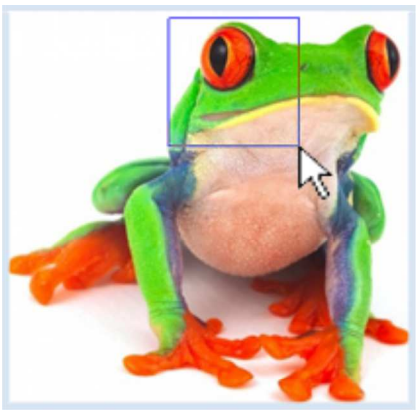
MEMENTO

Le code effectuant le traitement de l'image est contenu dans la fonction **doIt()**. Le reste du code n'est que réutilisation des programmes précédents. Le traitement se fait simplement en parcourant tous les pixels de la zone rectangulaire sélectionnée et en diminuant de 25% la **composante rouge**. Remarquez le double slash qui représente une division entière permettant d'obtenir à nouveau un nombre entier.
Le programme présente quelques bugs dérangeants facilement corrigeables. Premièrement, le traitement altère les zones qui ne sont pas rouges et fait par exemple virer le blanc vers le bleu. Deuxièmement, il plante lorsque le rectangle de sélection est tiré en dehors de la fenêtre.

Il serait évidemment très pratique que le programme détecte lui-même la zone à laquelle appliquer le filtre des yeux rouges. Réaliser une telle fonctionnalité demande cependant d'être un barbu du traitement d'images car il faut pour cela que le programme soit capable d'analyser l'image et d'effectuer une certaine reconnaissance de forme qui est un problème difficile en informatique [**plus...**].

COUPER ET STOCKER DES IMAGES

Découper des portions d'une image fait également partie des fonctions de base d'un programme de traitement d'images. Le programme suivant permet de sélectionner une zone rectangulaire de l'image qui sera ensuite copiée dans une autre fenêtre et enregistrée au format JPEG une fois le bouton de la souris relâché.



```
from gpanel import *

size = 300

def onMousePressed(e):
    global x1, y1
    global x2, y2
    setColor("blue")
    setXORMode("white")
    x1 = x2 = e.getX()
    y1 = y2 = e.getY()

def onMouseDragged(e):
    global x2, y2
    rectangle(x1, y1, x2, y2) # erase old
    x2 = e.getX()
```

```
    y2 = e.getY()
    rectangle(x1, y1, x2, y2) # draw new

def onMouseReleased(e):
    rectangle(x1, y1, x2, y2) # erase old
    setPaintMode()
    ulx = min(x1, x2)
    lrx = max(x1, x2)
    uly = min(y1, y2)
    lry = max(y1, y2)
    doIt(ulx, uly, lrx, lry)

def doIt(ulx, uly, lrx, lry):
    width = lrx - ulx
    height = lry - uly
    if ulx < 0 or uly < 0 or lrx > size or lry > size:
        return
    if width < 20 or height < 20:
        return

    cropped = GBitmap.crop(img, ulx, uly, lrx, lry)
    p = GPanel(Size(width, height))  # another GPanel
    p.window(0, width, 0, height)
    p.image(cropped, 0, 0)
    rc = save(cropped, "mypict.jpg", "jpg")
    if rc:
        p.title("Saving OK")
    else:
        p.title("Saving Failed")


x1 = y1 = 0
x2 = y2 = 0

makeGPanel(Size(size, size),
    mousePressed = onMousePressed,
    mouseDragged = onMouseDragged,
    mouseReleased = onMouseReleased)
window(0, size, size, 0)    # y axis downwards

img = getImage("sprites/colorfrog.png")
image(img, 0, size)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Il est possible d'afficher plusieurs fenêtres GPanel en créant plusieurs objets GPanel. Pour spécifier dans laquelle il faut effectuer les instructions de dessin, il faut invoquer les opérations graphiques avec l'opérateur *point*. Si la région sélectionnée est trop petite, par exemple si on clique avec la souris sans la déplacer, ou si elle se situe en partie en dehors de la fenêtre, la fonction *doIt()* se termine avec une instruction *return* vide.

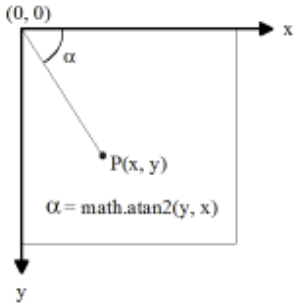
Pour sauvegarder une image, on peut utiliser la méthode **save()** où le dernier paramètre détermine le format de l'image. Les valeurs permises sont "*bmp*", "*gif*", "*jpg*", "*png*".

EXERCICES

1. Développer un programme qui inverse les composantes rouge et vert de *colorfrog.png*.

2. Développer un programme permettant de tourner l'image en glissant la souris. Utiliser la fonction *atan2(y, x) qui renvoie l'angle (en radians) entre le segment OP* et l'horizontale pour n'importe quel point *P(x, y)* du canevas. Il ne faut pas oublier de convertir les radians en degrés avec la fonction *degrees* du module *math* avant de pouvoir tourner l'image avec *GBitmap.scale()*.

   Prendre l'image *colorfrog.png* comme image de test.



3. Développer un programme de retouche de photos permettant de mémoriser la couleur du pixel survolé par la souris lors d'un clic (pipette de couleur). Ensuite, chaque glissé de la souris devrait dessiner un disque rempli de la couleur mémorisée, dont le centre correspond au premier clic et tel que le point auquel la souris est relâchée se trouve sur le bord du cercle. Il faudra faire usage des événements *press*, *drag*, et *click* events. Utiliser à nouveau *colorfrog.png* comme image de test. Écrire les trois composantes fondamentales de la couleur mémorisée dans la barre de titre de la fenêtre.

MATÉRIEL BONUS

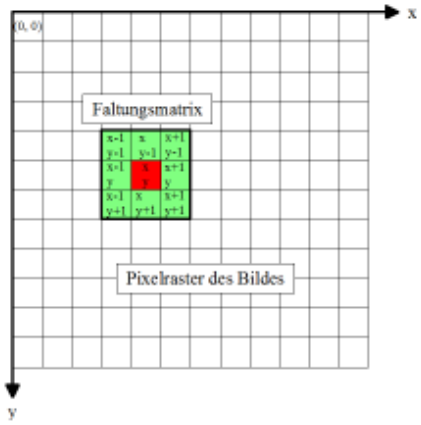
FILTRER DES IMAGES PAR CONVOLUTION

Vous connaissez certainement des programmes de traitement d'images mettant à disposition de nombreux filtres tels que le lissage, l'accentuation des contours, le floutage, etc … L'implémentation de ces filtres fait systématiquement appel à une opération mathématique appelée **convolution** sur laquelle vous pouvez sans problème vous renseigner plus en détails sur le Web [**plus...**]. In this process, you change the color values of each pixel by calculating a new value from it and its eight neighboring pixels, according to a filtering rule.

Voici une explication plus précise du fonctionnement de la convolution sur une image en nuances de gris où chaque pixel possède une valeur de gris *v* comprise entre 0 et 255. La règle de filtrage est définie par neuf nombres disposés en carré :

m00 m01 m02
m10 m11 m12
m20 m21 m22

Cette représentation est appelée **matrice de convolution** (ou **masque**). En *Python*, cette dernière est implémentée comme une liste de lignes de la matrice, chaque ligne étant elle-même représentée par une liste :



mask = [[0, -1, 0], [-1, 5, 1], [0, -1, 0]]

Cette structure de données offre un accès facile à chaque élément de la matrice de convolution avec un double indice. Par exemple m12 = mask[1][2] = 1. Ces neuf nombres sont utilisés comme des facteurs de pondération entre le pixel central en cours de traitement et ses huit voisins qui permettent de calculer la nouvelle valeur *vnew* que prendra le pixel en fonction des valeurs actuelles v(x, y) du pixel et de ses huit voisins. Le calcul est effectué de la manière suivante :

```
vnew(x, y) =   m00 * v(x - 1, y -1)   + m01 * v(x, y - 1)   + m02 * v(x + 1, y - 1) +
               m10 * v(x - 1, y)      + m11 * v(x, y)       + m12 * v(x + 1, y) +
               m20 * v(x - 1, y + 1)  + m21 * v(x , y + 1)  + m22 * v(x + 1, y + 1)
```

Pour simplifier, on pourrait dire que pour calculer la valeur de gris d'un pixel (rouge sur le schéma), on place le centre de la matrice de convolution au-dessus du pixel à recalculer, que l'on multiplie chacun de ses éléments avec la valeur de gris du pixel sous-jacent et que l'on fait la somme de ces neufs produits.
Le programme suivant effectue cette opération de convolution pour chacun des pixels excepté ceux qui se trouvent dans les bords et enregistre les valeurs de gris résultant dans un nouveau bitmap qui est ensuite affiché. Pour cela, la matrice de convolution est déplacée ligne par ligne, de gauche à droite et du haut vers le bas à l'intérieur d'une boucle *for*. La matrice de convolution utilisée correspond à un filtre d'accentuation de la netteté appliqué à l'image *frogbw.png de la grenouille.*

```
from gpanel import *

size = 300

makeGPanel(Size(2 * size, size))
window(0, size, size, 0)     # y axis downwards

bmIn = getImage("sprites/frogbw.png")
image(bmIn, 0, size)
w = bmIn.getWidth()
h = bmIn.getHeight()
bmOut = GBitmap(w, h)

#mask = [[1/9, 1/9, 1/9], [1/9, 1/9, 1/9], [1/9, 1/9, 1/9]]  # smoothing
mask = [[ 0, -1,  0], [-1,  5, -1], [0,  -1,  0]] #sharpening
#mask = [[-1, -2, -1], [ 0,  0,  0], [ 1,  2,  1]] #horizontal edge extraction
#mask = [[-1,  0,  1], [-2,  0,  2], [-1,  0,  1]] #vertical edge extraction

for x in range(0, w):
```

```
    for y in range(0, h):
        if x > 0 and x < w - 1 and y > 0 and y < h - 1:
            vnew = 0
            for k in range(3):
                for i in range(3):
                    c = bmIn.getPixelColor(x - 1 + i, y - 1 + k)
                    v = c.getRed()
                    vnew +=  v * mask[k][i]
            # Make int in 0..255
            vnew = int(vnew)
            vnew = max(vnew, 0)
            vnew = min(vnew, 255)
            gray = Color(vnew, vnew, vnew)
        else:
            c = bmIn.getPixelColor(x, y)
            v = c.getRed()
            gray = Color(v, v, v)

        bmOut.setPixelColor(x, y, gray)

image(bmOut, size / 2, size)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Lors d'une convolution, la valeur du pixel central est remplacée par une moyenne pondérée de la valeur actuelle du pixel et celle de ses huit voisins. La pondération des différents facteurs est indiquée par la matrice de convolution qui détermine le type de filtre dont il s'agit.

Pourquoi ne pas expérimenter avec les matrices de convolution bien connues ci-dessous ou inventer vos propres matrices de convolution ?

Type de filtre	Matrice de convolution
Filtre de netteté	$\begin{pmatrix} 0 & -1 & 0 \\ 1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}$
Filtre de lissage	$\begin{pmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{pmatrix}$
Filtre de détection de contours horizontal	$\begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$
Filtre de détection de contours vertical	$\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$

3.12 IMPRESSION D'IMAGES

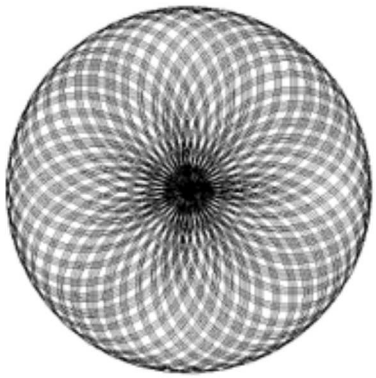
INTRODUCTION

Dans le chapitre portant sur les graphiques avec la tortue, vous avez déjà appris à faire en sorte que la tortue dessine sur une imprimante haute résolution au lieu de l'écran. De manière similaire, GPanel peut effectuer le rendu d'une image sur une imprimante papier ou une imprimante virtuelle telle qu'un fichier Tiff ou EPS. Pour effectuer une impression, il faut placer les instructions de dessin à l'intérieur d'une fonction dépourvue de paramètre, nommée par exemple *doIt()*. Lorsque cette fonction est appelée directement, le graphique résultant des instructions qu'elle exécute sera affiché à l'écran tandis qu'il sera envoyé vers l'imprimante en passant cette fonction *doIt* en paramètre à la fonction *printerPlot(doIt)*. Il est même possible de spécifier un facteur d'échelle *k* avec *printerPlot(doIt, k)* qui va causer un rétrécissement de l'image si $k < 1$ et un agrandissement si $k > 1$.

CONCEPTS DE PROGRAMMATION: *Graphiques haute résolution*

ROSACES

Les courbes à l'allure de roses remontent au XVIIIe siècle, plus précisément au mathématicien Guido Grandi [**plus...**] Les fonctions génératrices de ces courbes sont exprimées le plus naturellement en coordonnées polaires (ρ, φ) et sont paramétrées par un paramètre *n*:



$$\rho = \sin(n\varphi)$$

On passe aux coordonnées cartésiennes comme d'habitude avec les substitutions:

$$x = \rho \cos(\varphi)$$
$$y = \rho \sin(\varphi)$$

On obtient une jolie rosace en prenant n = $\sqrt{2}$. Elle sort encore bien mieux sur une imprimante qu'à l'écran.

```
from gpanel import *
import math

def rho(phi):
    return math.sin(n * phi)

def doIt():
    phi = 0
    while phi < nbTurns * math.pi:
        r = rho(phi)
        x = r * math.cos(phi)
        y = r * math.sin(phi)
        if phi == 0:
          move(x, y)
        else:
          draw(x, y)
        phi += dphi
```

```
n = math.sqrt(2)
dphi = 0.01
nbTurns = 100
makeGPanel(-1.2, 1.2, -1.2, 1.2)
doIt()
printerPlot(doIt)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

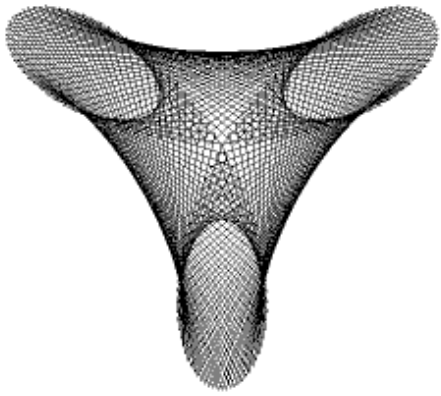
MEMENTO

Suivant le choix opéré pour le paramètre *n*, on peut créer différents types de courbes. N'hésitez pas à tester des valeurs entières, rationnelles et même irrationnelles telles que (π, e).

ROSES DE MAURER

Le mathématicien Peter Maurer a introduit ces courbes en 1987 dans son article « A Rose is a Rose... ». Elles se construisent à partir d'une rosace sur laquelle on choisit 360 points connectés entre eux par des segments droits. Les points de la rosace sélectionnés sont tous les points de la forme (sin(*nk*), *k*) pour *k* = 0, *d*, 2*d*, 3*d*, ..., 360*d* où *d* correspond à un angle de rotation exprimé en degrés.

Ces points sont ensuite reliés dans l'ordre par des segments droits.



Suivant le choix de n et d, des courbes aux allures totalement différentes vont être engendrées. Il faut les imprimer pour pouvoir les admirer dans toute leur beauté. L'exemple ci-contre peut être généré avec les valeurs n = 3 et d = 47°.

```
from gpanel import *
import math

def sin(x):
    return math.sin(math.radians(x))

def cos(x):
    return math.cos(math.radians(x))

def cartesian(polar):
    return [polar[0] * cos(polar[1]), polar[0] * sin(polar[1])]

def rho(phi):
    return sin(n * phi)

def doIt():
    for i in range(361):
        k = i * d
        pt = [rho(k), k]
        corners.append(pt)

    move(cartesian(corners[0]))
    for pt in corners:
        draw(cartesian(pt))

corners = []
```

```
n = 3
d = 47
makeGPanel(-1.2, 1.2, -1.2, 1.2)
doIt()
printerPlot(doIt)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Ce programme utilise des degrés et non des radians. De ce fait, il est pratique de définir ses propres fonctions sinus et cosinus qui prennent des angles en degrés. Cela simplifiera également le code qui ne nécessitera plus de spécifier explicitement le module *math* dans *math.sin* ou *math.cos* pour faire appel aux fonctions trigonométriques.
De manière similaire, il est pratique de définir une conversion des coordonnées polaires aux coordonnées cartésiennes dans la fonction *cartesian()* où les coordonnées sont passées sous forme de liste à deux éléments.

Le programme stocke les coordonnées polaires des 361 points de la rosace dans la liste *corners*. À la fin, on les parcourt dans l'ordre en les reliant avec la fonction *draw()*. D'autres roses de Maurer connues peuvent être obtenues avec les paramètres suivants :

n	d
2	39
2	31
6	71

EXERCICES

1. Dessiner 50 cercles concentriques de centre *center* avec la fonction *wave(center, wavelength)* où *wavelength* correspond à l'augmentation du rayon entre chaque cercle. On interprétera les cercles dessinés comme les crêtes d'une onde circulaire. Dessiner ensuite les ondes avec un centre légèrement décalé à chaque fois et observer les interférences apparaissant sur une version imprimée de l'image. Quelle courbe connue de la géométrie peut-on alors reconnaître ?

3.13 WIDGETS

INTRODUCTION

Les programmes que vous utilisez d'habitude sont dotés d'une interface utilisateur (GUI = Graphical User Interface). Parmi les éléments graphiques souvent présents, on compte les barres de menu, les champs de saisie ou les boutons. De tels composants graphiques sont appelés **widgets** et sont considérés comme des objets, tout comme les objets tortues que nous avions manipulés dans le chapitre *Objets tortues*. Si vous voulez développer un programme avec une interface utilisateur moderne, il est essentiel de connaître et bien comprendre les notions de base de la programmation orientée objets (POO = OOP en anglais, pour object-oriented programming) [**plus...**].

Les widgets sont répartis en différentes classes comme le montre la liste ci-dessous.

Widget	**Classe**
Boutons	JButton
Étiquettes	JLabel
Champs de texte	JTextField
Barres de menu	JMenuBar
Éléments de menu	JMenuItem
Menu comportant des sous-menus	JMenu

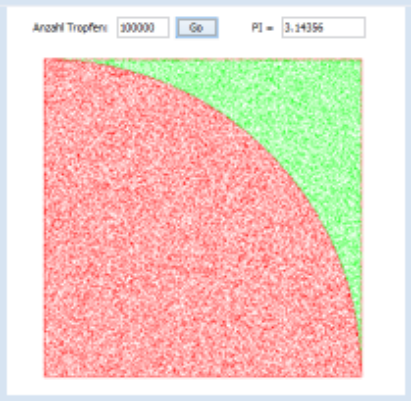
De même que l'on avait engendré une tortue en appelant le constructeur de la classe *turtle*, on doit créer un composant de l'interface utilisateur en appelant le constructeur de la classe idoine. Les constructeurs acceptent souvent des paramètres permettant d'initialiser certaines propriétés du widget. Par exemple, on peut créer un champ de saisie d'une longueur de 10 caractères avec *tf = JTextField(10)*.

Lors de l'appel du constructeur, il est également nécessaire de définir une variable qui servira ultérieurement à accéder à l'objet créé et retourné par l'appel au constructeur. Le code, *tf.getText()* retourne par exemple le texte présent dans le champ de texte référencé par la variable *tf*.

Pour rendre un widget visible dans le GPanel, on utilise la fonction *addComponent()* à laquelle on passe la variable référençant le widget à placer. Les widgets sont automatiquement placés dans l'ordre des appels à *addComponent()* dans la partie supérieure de la fenêtre GPanel [**plus...**].

CONCEPTS DE PROGRAMMATION: *Interface utilisateur graphique, Composant GUI, fonction de rappel*

ESTIMATION DE PI PAR SIMULATION MONTE CARLO



Nous avons appris comment déterminer l'aire de surfaces quelconques à l'aide d'une simulation Monte Carlo. Imaginons que l'on dessine un quart de disque de rayon 1 inscrit dans un carré de côté 1. Si l'on fait tomber de manière uniforme *n* gouttes de pluie sur le carré, on pourra facilement déterminer combien de gouttes tombent à l'intérieur du quart de disque en moyenne. Puisque l'aire du quart de disque est donnée par

$$S = \frac{1}{4} * r^2 * \pi = \frac{\pi}{4}$$

et que l'aire du carré vaut 1, le nombre de gouttes devrait être égal à

$$k = n * \frac{\pi}{4}$$

De ce fait, si l'on fait tomber n gouttes dans une simulation et que *k* gouttes tombent à l'intérieur du quart de disque, on obtient une approximation de pi avec

$$\pi = \frac{4 * k}{n}$$

L'interface utilisateur comporte deux étiquettes, deux champs de saisie textuels et un bouton. Une fois créés, on ajoute ces widgets au GPanel à l'aide de **addComponent()**.

Il est clair qu'un clic sur le bouton OK devrait être considéré comme un événement. La fonction de rappel qui sert de gestionnaire pour cet événement de clic est enregistrée par le paramètre **actionListener** dans le constructeur de *JButton*. Vous vous rappelez sûrement qu'il ne faut pas placer du code qui s'exécute sur une longue durée à l'intérieur des fonctions de rappel. De ce fait, on ne fait rien d'autre, dans la fonction de rappel, que d'appeler **wakeUp()** pour réveiller le programme qui avait été endormi dans la boucle *while* par l'appel à **putSleep()** dans le but de lancer la simulation.

```
from gpanel import *
import random
from javax.swing import *

def actionCallback(e):
    wakeUp()

def createGUI():
    addComponent(lbl1)
    addComponent(tf1)
    addComponent(btn1)
    addComponent(lbl2)
    addComponent(tf2)
    validate()

def init():
    tf2.setText("")
    clear()
    move(0.5, 0.5)
    rectangle(1, 1)
    move(0, 0)
    arc(1, 0, 90)

def doIt(n):
    hits = 0
```

```
    for i in range(n):
        zx = random.random()
        zy = random.random()
        if zx * zx + zy * zy < 1:
            hits = hits + 1
            setColor("red")
        else:
            setColor("green")
        point(zx, zy)
    return hits

lbl1 = JLabel("Number of drops: ")
lbl2 = JLabel("             PI = ")
tf1 = JTextField(6)
tf2 = JTextField(10)
btn1 = JButton("OK", actionListener = actionCallback)

makeGPanel("Monte Carlo Simulation", -0.1, 1.1, -0.1, 1.1)
createGUI()
tf1.setText("10000")
init()

while True:
    putSleep()
    init()
    n = int(tf1.getText())
    k = doIt(n)
    pi =  4 * k / n
    tf2.setText(str(pi))
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Les widgets sont des objets de la bibliothèque de classes Swing faisant partie de l'écosystème Java. Ils sont créés par l'appel du constructeur qui porte le même nom que la classe. Lors de l'appel du constructeur, on veille à créer une variable qui permet de référencer l'objet nouvellement créé pour une utilisation future. Pour afficher le widget dans GPanel, il faut appeler la fonction **addComponent()** en lui fournissant cette variable en tant que paramètre.

Une fois tous les widgets ajoutés au GPanel, il faut appeler la fonction **validate()**

pour lancer une régénération complète de la fenêtre. Ceci permet de tenir compte des nouveaux widgets insérés en garantissant un résultat tout-à-fait prévisible.

On peut enregistrer des gestionnaires d'événements avec le paramètre nommé **actionListener**. Il ne faut jamais exécuter du code long à terminer à l'intérieur d'une fonction de rappel.

LES MENUS (Rien à voir avec la cantine !)

De nombreuses fenêtres sont pourvues d'une barre de menu comportant plusieurs éléments de menu (menu items). Lors d'un clic sur un élément de menu, il n'est pas rare qu'un autre sous-menu s'ouvre à son tour, pouvant lui-même comporter plusieurs nouvelles entrées de menu. Les menus et entrées de menu sont également considérés comme des objets qu'il faut créer. La sélection d'une entrée de menu va également générer un événement qui sera traité par un gestionnaire d'événements approprié enregistré au préalable.

On construit une barre de menu en recourant à **JMenuBar()**et en ajoutant également un sous-menu. Il suffit pour cela de créer un objet *JMenu* et de lui ajouter un objet *JMenuItem*. Cela donnera lieu à un menu hiérarchique

Dans le but de simplifier un peu le code, on peut utiliser le même gestionnaire d'événement *actionCallback()* pour toutes les options du menu en le passant au paramètre **actionPerformed** lors de chaque appel au constructeur de **JMenuItem**. Dans le gestionnaire d'événements, on peut déterminer laquelle des entrées du menu a déclenché l'événement en invoquant la méthode **getSource()** de l'objet événement *e* reçu par le gestionnaire d'événement..

```
from gpanel import *
from javax.swing import *

def actionCallback(e):
    if e.getSource() == goItem:
        wakeUp()
    if e.getSource() == exitItem:
        dispose()
    if e.getSource() == aboutItem:
        msgDlg("Pyramides Version 1.0")

def doIt():
    clear()
    for i in range(1, 30):
        setColor(getRandomX11Color())
        fillRectangle(i/2, i - 0.35, 30 - i/2, i + 0.35)

fileMenu = JMenu("File")
goItem = JMenuItem("Go", actionPerformed = actionCallback)
exitItem = JMenuItem("Exit", actionPerformed = actionCallback)
fileMenu.add(goItem)
fileMenu.add(exitItem)

aboutItem = JMenuItem("About", actionPerformed = actionCallback)

menuBar = JMenuBar()
menuBar.add(fileMenu)
menuBar.add(aboutItem)

makeGPanel(menuBar, 0, 30, 0, 30)

while not isDisposed():
    putSleep()
    if not isDisposed():
        doIt()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

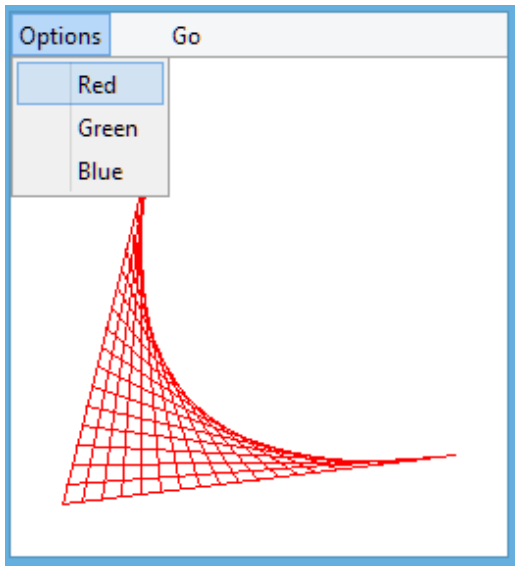
Rappelez-vous toujours qu'une fonction de rappel telle qu'un gestionnaire d'événements ne devrait jamais exécuter du code qui prend long à se terminer. On effectue de ce fait le dessin dans le programme principal.

Pour s'assurer que le programme se termine avec une certitude absolue lorsque l'utilisateur clique sur le bouton *fermer*, on utilise *isDisposed()* pour tester si la fenêtre a été fermée.

EXERCICES

1. Éditer le programme Moiré du chapitre 3.2 et ajouter une étiquette textuelle, un champ de saisie pour le réglage du délai et un bouton OK. Lors d'un clic sur le bouton OK, l'image sera rafraîchie avec le délai spécifié dans le champ de saisie (en millisecondes).

2. Éditer le programme de la section « algorithmes élégants d'art filaire » du chapitre 3.8 en y ajoutant le menu suivant : L'option de menu « Options » devrait contenir un sous-menu comportant les options de couleur « Rouge », « Vert », « Bleu ». L'option de menu « Go » devrait lancer le dessin du graphique filaire en utilisant la couleur sélectionnée dans le menu « Options ». Si aucune couleur n'est choisie dans le menu, le dessin sera fait en noir.



3*. Choisir un de vos graphiques favoris de ce chapitre pour le personnaliser en y ajoutant quelques widgets pour améliorer l'expérience utilisateur (UX = User eXperience).

Documentation GPanel

Importation du module: from gpanel import *

Fonction	Action
makeGPanel()	Crée une fenêtre de graphiques GPanel avec un système de coordonnées où x et y varient entre 0 et 1. Le curseur graphique est placé sur l'origine (0, 0) au coin inférieur gauche de la fenêtre
makeGPanel(xmin, xmax, ymin, ymax)	Crée une fenêtre GPanel avec les coordonnées flottantes indiquées. Le curseur graphique est placé en (0,0)
makeGPanel(xmin, xmax, ymin, ymax, False)	Idem, mais en cachant la fenêtre (pour la rendre visible, appeler visible(True))
makeGPanel(Size(width, height))	Idem que makeGPanel(), mais en spécifiant la taille de la fenêtre (en pixels)
getScreenWidth()	Retourne la largeur de l'écran (en pixels)
getScreenHeight()	Retourne la hauteur de l'écran (en pixels)
window(xmin, xmax, ymin, ymax)	Change la plage de coordonnées utilisées pour la fenêtre
drawGrid(x, y)	Dessiner une grille pour le système de coordonnées avec un marqueur tous les *x* pour l'axe horizontal et tous les *y* pour l'axe vertical. Les étiquettes des marqueurs sont déterminées par le type des paramètres *x* et *y* : *int* ou *float*
drawGrid(x, y, color)	Idem, en spécifiant la couleur *color* de la grille
drawGrid(x1, x2, y1, y2)	Idem, en indiquant la plage de coordonnées x1..x2, y1..y2 sur laquelle s'étend la grille
drawGrid(x1, x2, y1, y2, color)	Idem, en spécifiant la couleur *color* de la grille
drawGrid(x1, x2, y1, y2, x3, y3)	Idem, en spécifiant les unités utilisées pour la grille avec x3, y3 sur l'axe horizontal, respectivement vertical
drawGrid(x1, x2, y1, y2, x3, y3, color)	Idem, en spécifiant la couleur *color* de la grille
drawGrid(p, ...)	Idem que *drawGrid()* en utilisant le GPanel référencé par la variable p. Utile lors de l'utilisation conjointe de plusieurs panels
visible(isVisible)	Affiche / Cache la fenêtre
resizeable(isResizeable)	Spécifie si la fenêtre est redimensionnable. Valeur par défaut : non
dispose()	Ferme la fenêtre et libère les ressources utilisées pour le dessin
isDisposed()	Retourne *True* si la fenêtre est fermée par le bouton « fermer » de la barre de titre ou suite à l'appel de la fonction *dispose()*
bgColor(color)	Règle la couleur d'arrière-plan. Le paramètre *color* est une chaîne de couleur X11 ou un objet couleur retourné par le constructeur *makeColor()*
title(text)	Affiche *text* dans la barre de titre de la fenêtre
makeColor(colorStr)	Retourne un objet de type couleur correspondant à la chaine de couleur X11 passée en paramètre

windowPosition(ulx, uly)	Règle la position de la fenêtre par rapport aux coordonnées de l'écran (en pixels)
windowCenter()	Centre la fenêtre au milieu de l'écran
storeGraphis()	Stocke le graphique courant dans une mémoire tampon graphique
recallGraphics()	Effectue le rendu du contenu de la mémoire tampon dans le canevas GPanel
clearStore(color)	Efface la mémoire tampon graphique en y peignant avec la couleur indiquée
delay(time)	Met le programme en pause pour l'intervalle de temps indiqué par time (en ms)
getDividingPoint(pt1, pt2, ratio)	Retourne les coordonnées du point qui sépare le segment reliant les points *p1* et *p2* avec le rapport *ratio*. Le rapport peut être négatif ou supérieur à 1
getDividingPoint(c1, c2, ratio)	Idem, en utilisant les nombres complexes *c1* et *c2*
clear()	Réinitialise la fenêtre graphique en effaçant son contenu et en replaçant le curseur graphique à l'origine (0, 0)
erase()	Efface le contenu de la fenêtre graphique sans réinitialiser la position du curseur graphique
putSleep()	Met le programme en pause jusqu'à l'appel ultérieur de la fonction *wakeUp()*
wakeUp()	Remet le programme en route suite à une mise en pause avec *putSleep()*
linfit(X, Y)	Effectue une régression linéaire y = a*x + b sur la base des données contenues dans les listes X et Y et retourne les coefficients de la droite sous la forme du tuple *(a, b)*

Drawing

lineWidth(width)	Règle la largeur des lignes (en pixels)
setColor(color)	Règle la couleur de dessin à *color* (chaîne de couleur X11 ou objet de type *Color*)
move(x, y)	Place le curseur graphique à la position (x, y) sans dessiner de ligne
move(liste)	Idem, en spécifiant les coordonnées dans la liste *coord_list=[x, y]*
move(c)	Idem, en utilisant le nombre complexe *c* pour spécifier les coordonnées
getPosX()	Retourne la coordonnée *x* de la position du curseur
getPosY()	Retourne la coordonnée *y* de la position du curseur
getPos()	Retourne les coordonnées de la position du curseur sous forme de liste
draw(x, y)	Dessine une ligne depuis la position actuelle jusqu'au point *(x, y)* et met à jour la position du curseur avec *(x, y)*
draw(list)	Idem, en spécifiant les coordonnées dans une liste de la forme *[x, y]*
draw(c)	Idem, en spécifiant les coordonnées par le nombre complexe *c*
line(x1, y1, x2, y2)	Dessine une ligne de (x1, y1) vers (x2, y2) sans modifier la position actuelle du curseur

line(pt1, pt2)	Idem, en spécifiant les coordonnées des points de départ et d'arrivée de la ligne avec les listes *pt1 = [x1, y1]* et *pt2 = [x2, y2]*
line(c1, c2)	Idem, en spécifiant les coordonnées des points de départ et d'arrivée de la ligne par les nombres complexes *c1* et *c2*
circle(radius)	Dessine un cercle de rayon *radius* centré à la position actuelle du curseur graphique
fillCircle(radius)	Dessine un disque plein de rayon *radius* centré à la position actuelle du curseur graphique. La couleur de remplissage utilisée est déterminée par la couleur actuelle du pinceau
ellipse(a, b)	Dessine une ellipse vide d'axes *a* et *b* centrée à la position actuelle du curseur graphique
fillEllipse(a, b)	Dessine une ellipse pleine d'axes *a* et *b* centrée à la position actuelle du curseur graphique. La couleur de remplissage utilisée est déterminée par la couleur actuelle du pinceau
rectangle(a, b)	Dessine un rectangle de côtés *a* et *b* centré à la position actuelle du curseur graphique
rectangle(x1, y1, x2, y2)	Idem, en spécifiant le point supérieur gauche (x1, y1) et le point inférieur droit (x2, y2)
rectangle(pt1, pt2)	Idem, en spécifiant les points de la diagonale par les listes à deux éléments *p1* et *p2*
rectangle(c1, c2)	Idem, en utilisant les nombres complexes *c1* et *c2* pour spécifier les coordonnées
fillRectangle(a, b)	Dessine un rectangle plein de côtés *a* et *b* centré à la position actuelle du curseur graphique. La couleur de remplissage utilisée est déterminée par la couleur actuelle du pinceau
fillRrectangle(x1, y1, x2, y2)	Idem, en spécifiant le point supérieur gauche (x1, y1) et le point inférieur droit (x2, y2)
fillRectangle(pt1, pt2)	Idem, en spécifiant les points de la diagonale par les listes à deux éléments *p1* et *p2*
fillRrectangle(c1, c2)	Idem, en utilisant les nombres complexes *c1* et *c2* pour spécifier les coordonnées
arc(radius, startAngle, extendAngle)	Dessine un arc de cercle de rayon *radius* centré à la position du curseur et d'angle au centre *extendAngle*. *startAngle* indique l'angle du point de départ par rapport à l'horizontale. Le sens positif est le sens contraire des aiguilles de la montre
fillArc(radius, startAngle, extendAngle)	Idem, mais en dessinant un secteur circulaire plein. La couleur de remplissage utilisée est déterminée par la couleur actuelle du pinceau
polygon(x-list, y-list)	Dessine le polygone dont les coordonnées des sommets sont spécifiées par les listes *x_list*, respectivement *y_list*
polygon((li[pt1, pt2,..)	Idem en spécifiant les sommets dans une liste de points *pti* représentés par des listes de deux éléments *[x,y]*
polygon(li[c1, c2, c3,...])	Idem, en utilisant une liste de nombres complexes *c1, c2, c3*, … pour spécifier les coordonnées des sommets.

fillPolygon(x-list, y-list)	Dessine le polygone dont les coordonnées des sommets sont spécifiées par les listes *x_list*, respectivement *y_list*. La couleur de remplissage utilisée est déterminée par la couleur actuelle du pinceau
fillPolygon((li[pt1, pt2,..)	Idem en spécifiant les sommets dans une liste de points *pti* représentés par des listes de deux éléments *[x,y]*
fillPolygon(li[c1, c2, c3,...])	Idem, en utilisant une liste de nombres complexes *c1, c2, c3*, … pour spécifier les coordonnées des sommets
quadraticBezier(x1, y1, xc, yc, x1, y2)	Dessiner la courbe de Bézier quadratique définie par les points extrémaux *(x1, y1)* et *(x2, y2)* ainsi que le point de contrôle *(xc, yc)*
quadraticBezier(pt1, pc, pt2)	Idem en spécifiant les points avec des listes de deux éléments
quadraticBezier(c1, cc, c2)	Idem en spécifiant les points avec des nombres complexes
cubicBezier(x1, y1, xc1, yc1, xc2, yc2, x2, y2)	Dessiner la courbe de Bézier cubique définie par les points extrémaux *(x1, y1)* et *(x2, y2)* ainsi que les deux points de contrôle *(xc1, yc1)* et *(yc2, yc2)*
cubicBezier(pt1, ptc1, ptc2, pt2)	Idem en spécifiant les points avec des listes de deux éléments
cubicBezier(c1, cc1, cc2, c2)	Idem en spécifiant les points avec des nombres complexes
triangle(x1, y1, x2, y2, x3, y3)	Dessine un triangle de sommets (x1, y1), (x2, y2), (x3, y3)
triangle(pt1, pt2, pt3)	Idem en spécifiant les sommets avec des listes de deux éléments
triangle(c1, c2, c3)	Idem en spécifiant les sommets avec des nombres complexes
fillTriangle(x1, y1, x2, y2, x3, y3)	Dessine un triangle plein dont les sommets sont spécifiés par les points (x1, y1), (x2, y2), (x3, y3). La couleur de remplissage utilisée est déterminée par la couleur actuelle du pinceau
fillTriangle(pt1, pt2, pt3)	Idem en spécifiant les sommets avec des listes de deux éléments
fillTriangle(c1, c2, c3)	Idem en spécifiant les sommets avec des nombres complexes
point(x, y)	Dessine un point isolé (pixel) à la position *(x, y)*
point(pt)	Idem en spécifiant les coordonnées par une liste de deux éléments
point(complex)	Idem en spécifiant les coordonnées par un nombre complexe
fill(x, y, color, replacementColor)	Remplit la surface fermée contenant le point (x, y) en remplaçant tous les pixels actuellement de couleur *color* par un pixel de couleur *replacementColor*
fill(pt, color, replacementColor)	Idem, en spécifiant les coordonnées du point par la liste de deux éléments *pt*
fill(complex, color,replacementColor)	Idem, en spécifiant les coordonnées du point par le nombre complexe *complex*
image(path, x, y)	Insère l'image au format GIF, PNG ou JPEG stockée dans le fichier de chemin *path*. Place son coin inférieur gauche en (x,y). Le chemin *path* peut être relatif au dossier TigerJython, être contenu dans le dossier *sprites* de l'archive JAR de la distribution TigerJython ou être une URL débutant par *http://*
image(path, pt)	Idem, en spécifiant les coordonnées du coin inférieur gauche par la liste de deux éléments *pt*

image(path, complex)	Idem, en spécifiant les coordonnées du coin inférieur gauche par le nombre complexe *complex*
imageHeighpath)	Retourne la hauteur de l'image présente dans le fichier *path* (en pixels)
imageWidth(path)	Retourne la largeur de l'image présente dans le fichier *path* (en pixels)
enableRepaint(boolean)	Active/désactive le rendu automatique du tampon graphique. Valeur par défaut : *True* = activé
repaint()	Effectue le rendu du tampon graphique à l'écran. Nécessaire si le rendu automatique est désactivé
setPaintMode()	Active le mode de dessin standard qui dessine par-dessus l'arrière-plan
setXORMode(color)	Active le mode de dessin XOR qui effectue une combinaison XOR entre les pixels de l'arrière-plan et la couleur de dessin *color*. Deux dessins successifs identiques en mode XOR s'annulent pour redonner la couleur d'arrière-plan initiale
getPixelColor(x, y)	Retourne la couleur du pixel situé aux coordonnées (x, y) comme un objet de type *Color*
getPixelColor(pt)	Idem, pour le point dont les coordonnées sont indiquées dans la liste à deux éléments *pt*
getPixelColor(complex)	Idem, pour le point dont les coordonnées sont indiquées par le nombre complexe *complex*
getPixelColorStr(x, y)	Retourne la couleur du pixel situé aux coordonnées (x, y) comme une chaîne de couleur X11
getPixelColorStr(pt)	Idem, pour le point dont les coordonnées sont indiquées dans la liste à deux éléments *pt*
getPixelColorStr(complex)	Idem, pour le point dont les coordonnées sont indiquées par le nombre complexe *complex*

Texte

text(string)	Insère le texte contenu dans la chaîne *string* à partir de la position actuelle du curseur graphique
text(x, y, string)	Idem, en écrivant à partir du point de coordonnées (x, y)
text(pt, string)	Idem en spécifiant les coordonnées du point de départ par une liste de deux éléments
text(complex, string)	Idem en spécifiant les coordonnées du point de départ par un nombre complexe
text(x, y, string, font, textColor, bgColor)	Affiche le texte contenu dans *string* à partir de la position (x, y) en utilisant la police *font*, la couleur de texte *textColor* et la couleur d'arrière-fond *bgColor*
text(pt, string, font, textColor, bgColor)	Idem en spécifiant les coordonnées du point de départ par une liste de deux éléments
text(complex,string, font, textColor, bgColor)	Idem en spécifiant les coordonnées du point de départ par un nombre complexe
font(font)	Sélectionne une autre police pour les appels subséquents à *text()*

Fonctions de rappel

makeGPanel(mouseNNN = onMouseNNN) auch mehrere, durch Komma getrennt	Enregistre une fonction de rappel *onMouseNNN(x,y)* qui est appelée à chaque fois qu'un événement souris survient. Les valeurs possibles pour *NNN* sont: *Pressed, Released, Clicked, Dragged, Moved, Entered, Exited, SingleClicked, DoubleClicked*
isLeftMouseButton(), isRightMouseButton()	Retourne *True* si le dernier événement souris a été généré par un clique gauche, respectivement droit
makeGPanel(keyPressed = onKeyPressed)	Enregistre la fonction de rappel *onKeyPressed(keyCode)* qui est appelée
getKeyModifiers()	Retourne un nombre entier lorsque des touches spéciales du clavier sont enfoncées (Ctrl, Majuscule, Alt). Il est également possible d'obtenir ainsi les combinaisons de touches spéciales, par exemple Ctrl + Maj
makeGPanel(closeClicked = onCloseClicked)	Enregistre la fonction de rappel *onCloseClicked()* qui est appelée lorsque le bouton « fermer » de la barre des tâches est cliqué. On peut fermer la fenêtre à l'aide de l'appel *dispose()*

Keyboard

getKey()	Retourne, sous forme de chaîne de caractères, le caractère correspondant à la dernière touche du clavier enfoncée
getKeyCode()	Retourne le code (nombre entier) de la dernière touche du clavier enfoncée
getKeyWait()	Met le programme en pause jusqu'à ce qu'une touche du clavier soit actionnée et retourne le caractère en question sous forme de chaine de caractères
getKeyCodeWait()	Idem, en retournant le code de la dernière touche enfoncée sous forme de nombre entier
kbhit()	Retourne *True* si une touche du clavier a été pressée depuis le dernier appel à *getKey()* ou *getKeyCode()*

Composants d'interface graphique

add(component)	Insère un composant GUI vers le bord supérieur de la fenêtre
validate()	Redessine la fenêtre (et son contenu) après qu'un composant graphique a été rajouté avec *add()*

Barre d'état

addStatusBar(height)	Ajoute au bas de la fenêtre une barre d'état dont la hauteur est donnée par *height* (en pixels)
setStatusText(text)	Affiche le texte *text* dans la barre d'état. Le texte qui y figurait est supprimé
setStatusText(text, font, color)	Idem en indiquant la police et la couleur de texte à utiliser

Format de caractères

Font(name, style, size)	Crée un nouvel objet de type police de caractères utilisant la police nommée *name*, dans le style *style* et la taille de caractères *size.* Les différents paramètres ainsi que leur type sont décrits dans les trois lignes ci-dessous

name	Chaîne de caractères décrivant une police de caractères installée sur le système, comme par exemple "Times New Roman", "Arial" ou "Courier"
style	Nombre entier parmi les constantes suivantes: Font.PLAIN, Font.BOLD, Font.ITALIC. Ces constantes peuvent être combinées par addition, comme par exemple: Font.BOLD + Font.ITALIC
size	Nombre entier correspondant à une taille disponible pour la police de caractères choisie, par exemple 12, 16, 72

Boîtes de dialogue **Documentation EntryDialoque**

msgDlg(message)	Ouvre une boîte de dialogue modale comportant un bouton OK et le message *message*
msgDlg(message, title = title_text)	Idem, avec une barre de titre comportant le texte *title_text*
inputInt(prompt)	Ouvre une boîte de dialogue modale comportant des boutons OK et annuler. La fonction retourne le nombre entier saisi lors du clic sur OK ou interrompt le programme lors d'un clic sur annuler ou sur le bouton « fermer ». Si aucune valeur n'était présente lors de la validation ou s'il ne s'agissait pas d'un nombre entier, la boîte de dialogue réapparaît
inputInt(prompt, False)	Idem, excepté que les boutons annuler/fermer ne terminent pas le programme mais retournent la valeur *None*
inputFloat(prompt)	Ouvre une boîte de dialogue modale comportant des boutons OK et annuler. La fonction retourne le nombre flottant saisi lors du clic sur OK ou interrompt le programme lors d'un clic sur annuler ou sur le bouton « fermer ». Si aucune valeur n'était présente lors de la validation ou s'il ne s'agissait pas d'un nombre flottant, la boîte de dialogue réapparaît
inputFloat(prompt, False)	Idem, excepté que les boutons annuler/fermer ne terminent pas le programme mais retournent la valeur *None*
inputString(prompt)	Ouvre une boîte de dialogue modale comportant des boutons OK et annuler. La fonction retourne la chaine de caractères saisie lors du clic sur OK ou interrompt le programme lors d'un clic sur annuler ou sur le bouton « fermer »
inputString(prompt, False)	Idem, excepté que les boutons annuler/fermer ne terminent pas le programme mais retournent la valeur *None*
input(prompt)	Ouvre une boîte de dialogue modale comportant des boutons OK et annuler. La fonction retourne le nombre entier, le flottant ou, à défaut, la chaine de caractères saisie lors du clic sur OK ou interrompt le programme lors d'un clic sur annuler ou sur le bouton « fermer »
input(prompt, False)	Idem, excepté que les boutons annuler/fermer ne terminent pas le programme mais retournent la valeur *None*
askYesNo(prompt)	Ouvre une boîte de dialogue modale comportant les boutons oui/non. Le bouton « oui » retourne *True* et le bouton « non » retourne *False*. Le bouton « annuler » ou « fermer » terminent le programme
askYesNo(prompt, False)	Idem, excepté que les boutons annuler/fermer ne terminent pas le programme mais retournent la valeur *None*

Module import: from fitter import *

Curve fitting:

Function	Action
polynomfit(xdata, ydata, n)	fits a polynom of order n and returns the fitted values in ydata. Return value: list with n + 1 polynom coefficients
splinefit(xdata, ydata, nbKnots)	fits a spline function that passes through nbKnots aequidistant data points. Returns the fitted data in ydata
functionfit(func, derivatives, initialGuess, xdata, ydata)	fits the function func(x, param) with n parameters in list param. derivatives(x, param) returns a list with the values of the partial derivatives to the n parameters. initGuess is a list with n guessed values for the n parameters
functionfit(func, derivatives, initialGuess, xdata, ydata, weights)	same but with a list weights that determines the relative weights of the data points
toAequidistant(xrawdata, yrawdata, deltax)	returns two lists xdata, ydata with aequidistant values separated by deltax (linear interpolation)

TCP Client/Server Library:

Module import: from tcpcom import *

Class TCPServer

server = TCPServer(port, stateChanged, isVerbose = False)	creates a TCP socket server that listens on TCP port for a connecting client. State changes invoke the callback stateChanged(). For isVerbose = True, debug messages are written to the console
stateChanged(state, msg)	Callback called at state change events. state: TCPServer.PORT_IN_USE, msg: port state: TCPServer.CONNECTED, msg: IP address of client state: TCPServer.LISTENING, msg: empty state: TCPSever.TERMINATED, msg: empty state: TCPServer.MESSAGE, msg: message received from client (string)
server.isConnected()	True, if a client is connected to the server
server.disconnect()	closes the connection with the client and enters the the LISTENING state
server.terminate()	closes the connection and terminates the LISTENNG state. Releases the IP port
server.isTerminated()	True, if the server is in TERMINATED state
server.sendMessage(msg)	sends the information msg to the client (as string, the character \0 (ASCII 0) serves as end of string indicator, it is transparently added and removed)
TCPServer.getVersion()	returns the module version as string

Klasse TCPClient

client = TCPClient(ipAddress, port, stateChanged, isVerbose = False)	creates a TCP socket client prepared for a connection with a TCPServer at given address (string) and port (integer). State changes invoke the callback stateChanged(). For isVerbose = True, debug messages are written to the console

stateChanged(state, msg)	Callback called at state change events. state: TCPClient.CONNECTING, msg: IP address of server state: TCPClient.CONNECTION_FAILED, msg: IP address of server state: TCPClient.CONNECTED, msg: IP address of server state: TCPClient.DISCONNECTED, msg: empty state: TCPClient.MESSAGE, msg: message received from server (string)
client.connect()	creates a connection to the server (blocking until timeout). Returns True, if the connection is established; otherwise returns False
client.connect(timeout)	same, but with timeout (in s) to establish the connection
client.isConnecting()	True during a connection trial
client.isConnected()	True, if the client is connected to a server
client.disconnect()	closes the connection with the server
client.sendMessage(msg, responseTime)	sends the information msg to the server (as string, the character \0 (ASCII 0) serves as end of string indicator, it is transparently added and removed). For responseTime > 0 the method blocks and waits for maximum responseTime seconds for a server reply. Returns the message or None, if a timeout occured
TCPClient.getVersion()	returns the module version as string

chapitre quatre

SON

Objectifs d'apprentissage

- Être capable d'expliquer comment le son est digitalisé et codé.
- Comprendre le concept d'échantillonnage et connaître ses implications.
- Être capable d'enregistrer un son depuis un programme personnel, le modifier, le rejouer et le sauver dans un fichier.

"As the skills that constitute literacy evolve to accommodate digital media, computer science education finds itself in a sorry state. While students are more in need of computational skills than ever, computer science suffers dramatically low retention rates and a declining percentage of women and minorities. Studies of the problem point to the over-emphasis in computer science classes on abstraction over application, technical details instead of usability, and the stereotypical view of programmers as loners lacking creativity. Media Computation, teaches programming and computation in the context of media creation and manipulation."

In Forte, Guzdial, Not Calculation: Media as a Motivation and Context for Learning

4.1 RESTITUER UN SON

INTRODUCTION

Pour pouvoir traiter un signal sonore dans un ordinateur, il est nécessaire de le digitaliser au préalable. Pour ce faire, on utilise un convertisseur analogique-numérique pour l'échantillonner à intervalles de temps réguliers en convertissant la valeur mesurée à chaque instant d'échantillonnage en une valeur numérique. Le son est ainsi transformé en une suite de nombres que l'on peut stocker et traiter au sein de l'ordinateur. La fréquence d'échantillonnage (ou taux d'échantillonnage) est le nombre d'échantillons enregistrés par seconde. Pour le format audio WAV, les taux d'échantillonnages sont standardisés et figurent parmi les valeurs suivantes : 8000, 11025, 16000, 22050 et 44100 Hertz. Plus la fréquence d'échantillonnage est élevée, meilleure sera la qualité de la restauration sonore avec un convertisseur numérique-analogique. La plage de valeurs des échantillons est également d'une grande importance pour la qualité. Dans la bibliothèque sonore de *TigerJython*, les valeurs sont toujours stockées dans des listes de nombres entiers codés sur 16 bits, ce qui correspond à l'intervalle (-32768 and 32767).

Il faut également déterminer si l'on parle d'un son monaural (mono) ou binaural (stéréo) qui utilise un, respectivement deux canaux. Lors de l'utilisation de deux canaux (stéréo), les valeurs pour le canal de gauche et celui de droite sont stockées comme deux nombres consécutifs dans la liste d'échantillons.

Pour suivre ce chapitre, vous devrez vous équiper d'un ordinateur muni d'une carte son, d'écouteurs ou de haut-parleurs permettant de restituer les sons et d'un micro permettant d'effectuer des enregistrements.

CONCEPTS DE PROGRAMMATION: *Digitalisation du son, signal audio, échantillon, taux d'échantillonnage*

ÉCOUTER DES SONS

Pour profiter du programme suivant, il faut commencer par trouver sur Internet un petit clip sonore de quelques secondes au format WAV. Le clip sonore doit se situer dans un fichier nommé *mysound.wav* dans le même dossier que votre programme Python.

Dans le programme ci-dessous, on commence par importer toutes les fonctions de la **sound library**. On stocke ensuite dans la liste **samples** les échantillons sonores lus depuis le fichier et on affiche dans la **console window**, les informations de format du fichier audio puisqu'il est nécessaire de connaître le taux d'échantillonnage pour permettre une restitution correcte. Dans l'exemple ci-dessous, le taux d'échantillonnage est de **22050 Hz**. La fonction **openMonoPlayer** permet d'effectuer la restitution sonore. Si le son est rejoué avec un taux d'échantillonnage incorrect, il sera joué à une vitesse différente et, de ce fait, avec d'autres fréquences.

```
from soundsystem import *

samples = getWavMono("mysound.wav")
print getWavInfo("mysound.wav")

openMonoPlayer(samples, 22050)
play()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

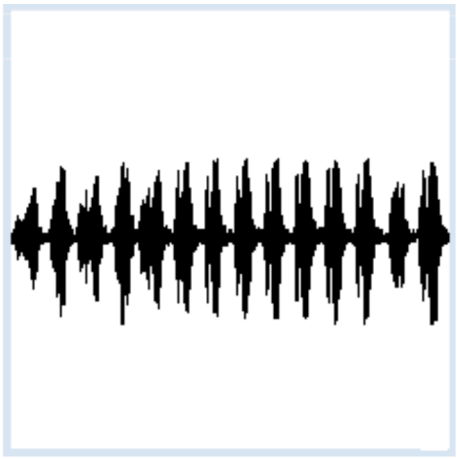
MEMENTO

La fonction **getWavMono()** permet de charger dans une liste les échantillons présents dans un fichier WAV. Chaque valeur lue sera un nombre entier compris entre -32768 et 32767. La fonction **openMonoPlayer(**) est un lecteur de sons permettant de jouer les sons avec la fonction *play()*.

Du fait que la longueur des listes est limitée par la capacité de la mémoire vive de l'ordinateur, la fonction *getWavMono()* ne permet de lire que des clips audio relativement courts.

ONDE SONORE

Il est intéressant de représenter graphiquement les échantillons sonores. Pour ce faire, on utilise simplement une fenêtre **GPanel**et l'on parcourt la liste des échantillons à l'aide d'une **for loop**.



```
from soundsystem import *

samples = getWavMono("mysound.wav")
print getWavInfo("mysound.wav")

openMonoPlayer(samples, 44100)
play()

from gpanel import *

makeGPanel(0, len(samples), -33000, 33000)
for i in range(len(samples)):
    draw(i, samples[i])
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Il est nécessaire de choisir le système de coordonnées du *GPanel* de manière appropriée. Les valeurs affichées dans la direction de l'axe *Ox* sont comprises entre 0 et le nombre d'échantillons, déterminable à partir de la longueur de la liste d'échantillons. Les valeurs de l'axe *Oy* sont comprises entre -32768 et 32767, ce qui explique que l'on utilise une plage de plus ou moins 33000.

LA VOIE DE LA FACILITÉ

Dans le cas où l'on ne s'intéresse qu'à jouer un fichier son, il suffit de trois lignes de code qui permettent d'ailleurs même de jouer des sons de longue durée, comme votre chanson favorite.

```
from soundsystem import *

openSoundPlayer("myfavoritesong.wav")
play()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

La distribution *TigerJython* met également à disposition une bibliothèque de plusieurs clips sonores dont les noms sont listés dans la table ci-dessous :

Fichier son	**Description**
wav/bird.wav	Chant d'oiseau
wav/boing.wav	Rebond
wav/cat.wav	Miaulement (cri) de chat
wav/click.wav	Clic de souris
wav/dummy.wav	Son vide
wav/explode.wav	Explosion
wav/frog.wav	Coassement de grenouille
wav/mmm.wav	Meuglement de vache
wav/notify.wav	Son de notification
wav/ping.wav	Bing ressemblant comme deux gouttes d'eau à celui de MS Windows

Cette liste de clips sonores est constamment étoffée. S'il s'avère qu'il existe dans le dossier de votre programme un fichier portant l'un des noms listés ci-dessus, il aura la priorité par rapport aux clips de *TigerJython*.

Le lecteur sonore dispose de nombreuses commandes de contrôle, à la manière des lecteurs professionnels. Il est par exemple possible de mettre la lecture en pause avec la fonction *pause()* et de la reprendre par la suite avec la fonction *play()*.

La longueur du son restitué n'est pas limitée avec ces fonctions car il n'est pas chargé en entier dans la mémoire de l'ordinateur. Bien au contraire, il est lu en streaming, à savoir par petits paquets.

play()	Lit un son depuis la position de lecture courante et retourne immédiatement
blockingPlay()	Idem mais en attendant que la lecture soit terminée pour retourner (fonction bloquante)
advanceFrames(n)	Saute depuis la position actuelle en avançant de *n* échantillons
advanceTime(t)	Depuis la position de lecture actuelle, avance de *t* millisecondes
getCurrentPos()	Indique la position courante de lecture au sein de la liste d'échantillons
getCurrentTime()	Indique la position temporelle actuelle de la lecture
pause()	Met la lecture en pause. La fonction *play()* permet de la relancer
rewindFrames(n)	Rembobine la lecture de *n* échantillons à partir de la position de lecture courante
rewindTime(t)	Idem, mais en reculant de *t* millisecondes
stop()	Arrête la lecture. La position de lecture est réinitialisée au début de l'enregistrement.
setVolume(v)	Ajuste le volume (valeur comprise entre 0 et 1000)

JOUER DES FICHIERS MP3

Pour jouer des fichiers audio au format MP3, il faut installer des bibliothèques supplémentaires disponibles en téléchargement depuis **ce lien**. Dézipper l'archive téléchargée dans le dossier *Lib* qu'il faut créer dans le même dossier que l'archive *tigerjython2.jar* s'il n'existe pas encore.

Pour jouer des fichiers MP3, il faut utiliser les fonctions **openSoundPlayerMP3(),** *openMonoPlayerMP3()* et *openStereoPlayerMP3* en lieu et place de *openSoundPlayer()*, *openMonoPlayer()* et *openStereoPlayer()* en indiquant le chemin d'accès au fichier MP3. Pour contrôler la lecture, il faut utiliser les mêmes fonctions que pour les fichiers WAV.

```
from soundsystem import *

openSoundPlayerMP3("song.mp3")
play()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Pour jouer des fichiers MP3, il faut se procurer des archives JAR additionnelles qu'il faut placer dans le dossier *Lib* du dossier racine de *tigerjyton2.jar*.

EXERCICES

1. Expliquer pourquoi les fréquences sonores sont altérées si l'on change le taux d'échantillonnage indiqué lors de la lecture.

2. Dans *GPanel*, montrer une onde sonore représentant une plage très courte (0.1 seconde) en faisant débuter la lecture à partir de 1 seconde. Expliquer l'image.

3. Créer un lecteur sonore disposant d'un *GPanel* et permettant d'exécuter les commandes suivantes au clavier :

Touche	**Action**
Flèche haut	Lecture
Flèche bas	Pause
Flèche gauche	Rembobinage de 10 s
Flèche droite	Avance rapide de 10 s
Touche S	Arrêt de la lecture

   Afficher la liste des commandes disponibles dans la fenêtre *GPanel*. Lors de chaque pression d'une touche, écrire la commande exécutée dans la barre de titre de la fenêtre.

4.2 ÉDITION DE SON

INTRODUCTION

Comme vous le savez, les échantillons sonores (les valeurs d'échantillonnage du son) sont stockés dans une liste et peuvent être restitués à partir de cette liste. Si l'on veut éditer le son, il suffit donc de modifier de façon tout-à-fait habituelle la liste des échantillons.

CONCEPTS DE PROGRAMMATION: *Onde rectangulaire, division entière, opérateur moduloon*

RÉGLER LE VOLUME DE L'ENREGISTREMENT

Le programme suivant réduit le volume à 25% du volume initial. Pour ce faire, on copie la liste des échantillons vers une **another list**, dont toutes les valeurs sont le quart de la valeur initiale.

```
from soundsystem import *

samples = getWavMono("mysound.wav")
soundlist = []
for item in samples:
    soundlist.append(item // 4)

openMonoPlayer(soundlist, 22010)
play()
```

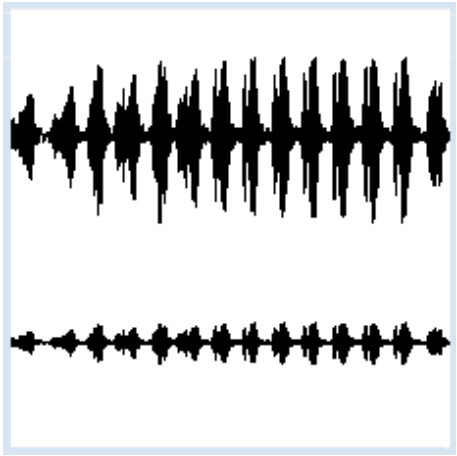
Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Pour copier une liste, il faut tout d'abord créer une **empty list**et la remplir petit à petit à l'aide de **append()**. Afin de s'assurer que les valeurs, une fois divisées par quatre, sont encore des nombres entiers, il faut utiliser la **integer division** (double slash de division).

UTILISER L'INDICE DE LA LISTE

Dans l'exemple suivant, on parcourt la liste à l'aide de **l'indice de liste** et l'on **hmodifier ses éléments** sans créer de nouvelle liste. L'onde sonore est représentée graphiquement avant et après l'opération.



```
from soundsystem import *
from gpanel import *
```

```
samples = getWavMono("mysound.wav")

makeGPanel(0, len(samples), -33000, 33000)
for i in range(len(samples)):
    if i == 0:
        move(i, samples[i] + 10000)
    else:
        draw(i, samples[i] + 10000)

for i in range(len(samples)):
    samples[i] = samples[i] // 4

for i in range(len(samples)):
    if i == 0:
        move(i, samples[i] - 10000)
    else:
        draw(i, samples[i] - 10000)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Beaucoup de programmeurs ont l'habitude de nommer *i* la variable qui contient l'indice de la liste. Ainsi, pour parcourir une liste, on utilise un bloc *for* de la manière suivante :

```
for i in range(10):
```

La variable *i* est aussi appelée « variable de contrôle » de la boucle ou « pas » de la boucle.

GÉNÉRER DES SONS

Il est assez grisant de pouvoir créer son propre son, non en le chargeant depuis un fichier préexistant, mais en créant la liste des échantillons de toute pièce. Pour créer une onde rectangulaire, il suffit de stocker une même valeur arbitraire, par exemple 5000, sur un nombre *N* d'échantillons et de poursuivre ensuite avec la valeur opposée, en l'occurrence -5000, sur les *N* échantillons suivants au sein de la liste.

```
from soundsystem import *

samples = []
for i in range(4 * 5000):
   value = 5000
   if i % 10 == 0:
       value = -value
   samples.append(value)

openMonoPlayer(samples, 5000)
play()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Le taux d'échantillonnage de 10000 Hz correspond à la prise d'un échantillon sonore tous les 0.1 ms. Le programme change toujours de signe (-/+) toutes les 10 valeurs, à savoir tous les 1 ms. Cela correspond à une onde rectangulaire dont la période est de 2 ms, ce qui correspond à une fréquence de 500 Hz. On utilise pour ce faire l'opérateur modulo % qui retourne le reste de la division entière. La condition **i % 10 == 0** est donc vraie pour i = 0, 10, 20, 30, etc.

EXERCICES

1. Utiliser l'opération de liste *reverse()* pour jouer le son à l'envers. L'effet est particulièrement saisissant avec des enregistrements de texte parlé.

2. L'opérateur de slicing *list[start: end]* permet de créer une nouvelle liste ne contenant que les éléments de la liste originale situés entre l'indice *start* (compris) et *end* (non compris). En utilisant cette astuce, supprimer une partie des échantillons d'un que vous avez sous la main.

3. Charger un clip sonore et déterminer la valeur maximale de son amplitude. Afficher cette valeur dans la barre de titre de la fenêtre *GPanel* et afficher une représentation graphique de l'onde sonore. Augmenter ensuite toutes les valeurs d'échantillonnage proportionnellement de telle manière que la valeur maximale atteigne le volume maximal possible, à savoir 32767, et représenter la nouvelle onde ainsi obtenue. Il s'agit d'une fonction essentielle de tout logiciel de traitement sonore appelée « normalisation ».

4*. Générer une onde sonore d'environ 500 Hz à un taux d'échantillonnage de 10000 Hz dont la forme est sinusoïdale. Utiliser à cet effet la fonction *math.sin(x)* qui est périodique de période x = 2π = 6.28. N'oubliez pas d'ajouter *import math* pour avoir accès à la fonction *math.sin*.

5*. Superposer par addition des échantillons deux ondes sinusoïdales dont les fréquences sont très proches. Quelle observation acoustique peut-on faire en jouant le son ?

4.3 ENREGISTRER DES SONS

INTRODUCTION

Il est également possible d'utiliser le système sonore pour enregistrer et sauvegarder des sons. Il faut pour cela commencer par connecter une source sonore telle qu'un microphone ou un appareil de lecture audio à l'entrée de la carte son. Les ordinateurs portables disposent en général d'un microphone rudimentaire intégré.

CONCEPTS DE PROGRAMMATION: Fonctions bloquantes et non bloquantes

ENREGISTREUR DE SONS

Avant d'effectuer un enregistrement, il faut, pour préparer le système d'enregistrement, appeler la fonction **openMonoRecorder()** en spécifiant le taux d'échantillonnage en guise de premier paramètre. On peut ensuite démarrer l'enregistrement à l'aide de la fonction **capture()** qui est une fonction non bloquante retournant immédiatement.

Pour terminer l'enregistrement, il faut appeler la fonction **stopCapture().** Les échantillons ainsi enregistrés sont copiés dans une liste qu'il est possible de récupérer à l'aide de **getCapturedSound().** Dans le programme suivant, on effectue un enregistrement de 5 secondes qui est ensuite rejoué immédiatement.

```
from soundsystem import *

openMonoRecorder(22050)
print("Recording...");
capture()
delay(5000)
stopCapture()
print("Stopped");
sound = getCapturedSound()

openMonoPlayer(sound, 22050)
play()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Une commande telle que **capture()**, qui déclenche une action et retourne immédiatement est appelée « **fonction non bloquante** ». De telles fonctions permettent de contrôler depuis le programme des tâches de fond qui se déroulent indépendamment de l'exécution du programme principal. Cela permet par exemple de les interrompre en cours d'exécution.

REPRÉSENTATION SONORE DU SON ENREGISTRÉ

On s'intéresse très souvent à la représentation graphique du son enregistré. Vous devriez normalement déjà savoir comment réaliser cette représentation graphique à l'aide de *GPanel*.
Le graphique adjacent représente l'enregistrement des mots
"one two three for five six seven eight nine ten".

```
from soundsystem import *

openMonoRecorder(22050)
print("Recording...");
capture()
delay(5000)
stopCapture()
print("Stopped");
sound = getCapturedSound()

from gpanel import *
makeGPanel(0, len(sound), -33000, 33000)
for i in range(len(sound)):
    draw(i, sound[i])
```

MEMENTO

On peut utiliser la longueur de la liste des échantillons pour déterminer le nombre d'échantillons enregistrés. Pour dessiner la représentation graphique il suffit d'utiliser une structure *for*. Amusez-vous quelques minutes en représentant graphiquement différents enregistrements sonores et assurez-vous de bien comprendre ce que représente la courbe du son.

ENREGISTREMENT DE FICHIERS WAV

Il est également possible de sauver le son enregistré dans un fichier WAV à l'aide de la fonction **writeWavFile()**.

```
from soundsystem import *

openMonoRecorder(22050)
print("Recording...");
capture()
delay(5000)
stopCapture()
print("Stopped");
sound = getCapturedSound()

writeWavFile(sound, "mysound.wav")
```

MEMENTO

Après avoir enregistré le son dans un fichier, il est possible de le rejouer à l'aide d'un programme Python ou de tout autre lecteur multimédia installé sur votre machine.

EXERCICES

1. Enregistrer des mots individuels

2. Arranger les mots précédemment enregistrés pour construire une phrase.

3*. Développer un programme qui prend un numéro de téléphone en entrée, en tant que chaine de caractères, et utilise des enregistrements sonores des différents chiffres pour prononcer le numéro de téléphone lu en entrée.

4.4 SYNTHÈSE VOCALE

INTRODUCTION

La synthèse vocale consiste à faire parler l'ordinateur avec une voix humaine. Un système de synthèse vocale (TTS = text-to-speech) convertit du texte en une sortie vocal. Prononcer un texte avec une voix synthétique qui semble naturelle est un problème très complexe dans lequel des avancées considérables ont été réalisées ces dernières années. Par rapport à la restitution d'un texte par lectures successives de clips sonores préenregistrés pour chacun des mots, les systèmes de synthèse vocale présentent l'avantage d'être très flexibles et de pouvoir lire n'importe quel texte. La synthèse vocale fait partie du domaine de la linguistique informatique. Le développement d'un système de synthèse vocale demande de ce fait une collaboration étroite entre linguistes et informaticiens.

Le système de synthèse vocale utilisé dans *TigerJython* est appelé *MaryTTS*et est développé au département de linguistique informatique et phonétique de l'université de Saarland en Allemagne.

Ce système utilise de très grosses bibliothèques sonores qu'il est nécessaire de télécharger séparément depuis **ce lien** et de décompresser dans le sous-dossier *Lib* du dossier dans lequel se trouve l'archive *tigerjython2.jar*.

CONCEPTS DE PROGRAMMATION: *Langage artificiel, synthèse vocale*

PARLER UN TEXTE EN 4 LANGUES

Les versions récentes de MaryTTS sont capables de synthétiser de nombreuses langues différentes, en particulier le français, l'allemand, l'italien et l'anglais. Pour chacune des langues, on peut choisir une voix féminine ou masculine à l'aide de la fonction **selectVoice()**. Il est ensuite possible d'appeler la fonction **generateVoice()** en lui transmettant le texte à prononcer sous forme de chaîne de caractères. Cette fonction retourne une liste contenant les échantillons sonores générés qu'il est possible de jouer à l'aide du lecteur audio.

```
from soundsystem import *

initTTS()

selectVoice("german-man")
#selectVoice("german-woman")
#selectVoice("english-man")
#selectVoice("english-woman")
#selectVoice("french-woman")
#selectVoice("french-man")
#selectVoice("italian-woman")

text = "Danke dass du mir eine Sprache gibst. Viel Spass beim Programmieren"
#text = "Thank you to give me a voice. Enjoy programming"
#text = "Merci pour me donner une voix. Profitez de la programmation"
#text = "Grazie che tu mi dia una lingua. Godere della programmazione"
voice = generateVoice(text)
openSoundPlayer(voice)
play()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Amusez-vous à lire le texte à l'aide des différentes voix en commentant / décommentant les différentes lignes. Il est toujours nécessaire de commencer par appeler la fonction **initTTS()** pour initialiser le système de synthèse vocale.

Il est également possible d'indiquer à la fonction *initTTS()* le chemin du dossier contenant les fichiers de données de *MaryTTS*. Par défaut, il s'agit du sous-dossier *Lib*.

ANNONCER LA DATE ET L'HEURE ACTUELLE

On peut citer de nombreuses applications de la synthèse locale. L'ordinateur peut par exemple prononcer des textes pour les personnes mal voyantes. Les annonces faites dans les trains ou les gares sont généralement des voix synthétiques. De nombreux jeux vidéo interactifs reposent sur des voix artificielles. Le programme suivant détermine l'heure actuelle en interrogeant le système d'exploitation puis l'annonce à haute voix, en français, allemand ou anglais.

```
from soundsystem import *
import datetime

language = "german"
#language = "english"

initTTS()
if language == "german":
    selectVoice("german-woman")
    month = ["Januar", "Februar", "März", "April", "Mai",
        "Juni", "Juli", "August", "September", "Oktober",
        "November", "Dezember"]
if language == "english":
    selectVoice("english-man")
    month = ["January", "February", "March", "April", "May",
        "June", "July", "August", "September", "October",
        "November", "December"]

now = datetime.datetime.now()

if language == "german":
    text = "Heute ist der " + str(now.day) + ". " \
        + month[now.month - 1] + " " + str(now.year) + ".\n" \
        + "Die genaue Zeit ist " + str(now.hour) + " Uhr " + str(now.minute)
if language == "english":
    text = "Today we have  " + month[now.month - 1] + " "  \
        + str(now.day) + ", "+ str(now.year) + ".\n" \
        + "The time is " + str(now.hour) + " hours " + str(now.minute)
        + " minutes."

print text
voice = generateVoice(text)
openSoundPlayer(voice)
play()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMO

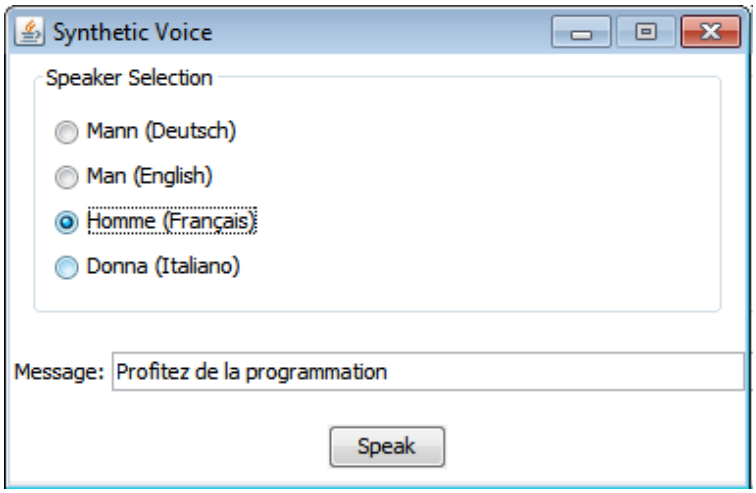
Les commentaires permettent de sélectionner la langue de lecture. La classe **datetime.datetime.now()** fournit des informations concernant l'heure et la date actuelles grâce aux attributs *year, month, day, hour, minute, second,* et *microsecond*.

Comme vous pouvez l'observer, il est possible d'étendre une expression Python sur plusieurs lignes en utilisant des caractères de backslash. Cette technique est particulièrement utile pour écrire de longues concaténations de chaines de caractères de manière plus lisible.

CRÉER SA PROPRE INTERFACE GRAPHIQUE

Comme nous l'avons déjà vu au chapitre 3.13, il est relativement simple de créer une boîte de dialogue élémentaire à l'aide de la classe *EntryDialog* de *TigerJython*. Comme c'est le cas dans de très nombreux environnements de programmation, les contrôles classiques tels que les champs textuels, les boutons, les boutons à cocher, les boutons radio ainsi que les curseurs sont modélisés par des objets logiciels qui apparaissent dans un panneau rectangulaire. La boîte de dialogue reste ouverte tant que le programme s'exécute (boîte de dialogue non modale). Pour de plus amples informations, consulter la documentation APLU.

Le programme ouvre une boîte de dialogue non modale permettant de sélectionner la langue de lecture. Lors du clic sur le bouton de confirmation, le texte figurant dans le champ textuel est prononcé par la voix sélectionnée.




```
from soundsystem import *
from entrydialog import *

speaker1 = RadioEntry("Mann (Deutsch)")
speaker1.setValue(True)
speaker2 = RadioEntry("Man (English)")
speaker3 = RadioEntry("Homme (Français)")
speaker4 = RadioEntry("Donna (Italiano)")
pane1 = EntryPane("Speaker Selection",
                  speaker1, speaker2, speaker3, speaker4)
textEntry = StringEntry("Message:", "Viel Spass am Programmieren")
pane2 = EntryPane(textEntry)
okButton = ButtonEntry("Speak")
pane3 = EntryPane(okButton)
dlg = EntryDialog(pane1, pane2, pane3)
dlg.setTitle("Synthetic Voice")

initTTS()

while not dlg.isDisposed():
    if speaker1.isTouched():
        textEntry.setValue("Viel Spass am Programmieren")
    elif speaker2.isTouched():
        textEntry.setValue("Enjoy programming")
    elif speaker3.isTouched():
        textEntry.setValue("Profitez de la programmation")
    elif speaker4.isTouched():
        textEntry.setValue("Godere della programmazione")
```

```
    if okButton.isTouched():
        if speaker1.getValue():
            selectVoice("german-man")
            text = textEntry.getValue()
        elif speaker2.getValue():
            selectVoice("english-man")
            text = textEntry.getValue()
        elif speaker3.getValue():
            selectVoice("french-man")
            text = textEntry.getValue()
        elif speaker4.getValue():
            selectVoice("italian-woman")
            text = textEntry.getValue()
        if text != "":
            voice = generateVoice(text)
            openSoundPlayer(voice)
            play()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

La boucle *while* s'exécute jusqu'à ce que la boîte de dialogue soit fermée à l'aide du bouton « fermer » de la barre de titre. Lors de chaque cycle de la boucle, la fonction **isTouched()** permet de déterminer si le bouton de confirmation a été cliqué depuis le dernier appel de la fonction. Si tel est le cas, on récupère la valeur des éléments graphiques (champ textuel et boutons radio) en appelant la fonction **getValue()** et en prononçant la chaine de caractères contenue dans le champ textuel à l'aide du système de synthèse vocale, exactement comme dans les exemples précédents.

Il est dangereux de laisser l'ordinateur foncer tête baissée dans une boucle aussi « serrée » ne faisant pratiquement rien d'autre que de tester si le bouton a été enfoncé ou non. Heureusement, l'appel de la fonction *isTouched()* met automatiquement le programme en pause pour une courte période (1 milliseconde) de telle sorte que le parcours de la boucle ne soit pas inutilement trop fréquent.

EXERCICES

1. Écrire un poème français dans un fichier texte, par exemple :

   Autrefois tout le monde croyait aux dragons, Jack Prelutsky (**poème original**)

   Autrefois tout le monde croyait aux dragons,
   Quand le monde était tout neuf et beau,
   On nous mettait dans des légendes,
   On faisait des récits, on chantait des chansons,
   On nous traitait avec respect,
   Les gens nous honoraient, les gens nous redoutaient,
   Mais un jour ils ont cessé d'y croire
   Depuis ce jour, nous avons disparu.
   Aujourd'hui, on dit que nous avons fait notre temps,
   que nous avons vécu,
   On nous traite avec dérision
   Nous qui, autrefois, étions des rois.
   Mais il faudra bien qu'un jour ils se souviennent,
   Que, d'une façon ou d'une autre, on leur révèle
   Que notre esprit est éternel
   Nous sommes des dragons ! Nous existons !

Jack Prelutsky (**Télécharger)**

La ligne line *text = open("dragon.txt", "r").read()* permet de lire le texte depuis le fichier texte *dragon.txt* situé dans le même dossier que le programme et de le stocker dans la chaine de caractères *text*. Faire en sorte que le texte soit lu en français.

2. Définir de manière récursive ou itérative la fonction *fac(n)* retournant la factorielle de *n*

   n! = 1 * 2 * ... *n

   Votre programme doit demander à l'utilisateur un nombre entre 1 et 10 à l'aide de *readInt()* en prononçant également la question à haute voix en français. Le programme doit ensuite calculer la factorielle *n!* du nombre saisi et annoncer le résultat à haute voix.

4.5 EXPÉRIENCES ACOUSTIQUES

INTRODUCTION

Il est également possible d'utiliser l'ordinateur pour remplacer un système expérimental onéreux. On pourrait par exemple imaginer faire des recherches sur l'ouïe humaine en utilisant le système sonore de TigerJython, ce qui est bien meilleur marché et procure une flexibilité énorme lorsque l'on conduit l'expérience à l'aide d'un programme « maison ».

CONCEPTS DE PROGRAMMATION: *Fréquence de référence, battements, gamme musicale*

ACCORDER UN INSTRUMENT DE MUSIQUE, BATTEMENTS

L'ouïe humaine ne peut pas distinguer deux sons de fréquences très proches lorsqu'ils sont joués séparément. En revanche, si les deux sons sont joués simultanément, il se produit un phénomène de battement caractérisé par des augmentations et des diminutions de volume qui sont très clairement audibles. Pour bien entendre ce phénomène, le programme suivant commence par jouer deux sons de fréquences très proches. Le premier est un joué à 440 Hz et le deuxième à 441 Hz. Lorsque les deux sons sont joués séparément, seule une oreille très développée percevra la différence. En revanche, lorsque les deux sons sont joués ensemble, on entend très clairement le phénomène de battement.

```
import time

playTone(440, 5000)
time.sleep(2)
playTone(441, 5000)
time.sleep(2)
playTone(440, 20000, block = False)
playTone(441, 20000)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

La fonction globale *playTone()* accepte différentes variantes d'arguments qui sont exposées dans l'aide de *TigerJython* sous la rubrique « Documentation APLU (Fenêtre graphique et autres gadgets) ». Dans le précédent programme, nous avons fait usage du paramètre *block* qui permet d'indiquer si la fonction doit être bloquante (ne retourner qu'après la fin du son) ou si elle doit être non bloquante (retourner immédiatement, sans attendre la fin du son). Il est nécessaire d'utiliser la version non bloquante pour jouer plusieurs sons simultanément.

Pour accorder des instruments au sein d'un orchestre et même pour des instruments seuls (instrument à corde, piano, etc.), on joue deux notes simultanément tout en prêtant attention aux battements qui surviennent lorsque les notes sont jouées à des fréquences très proches mais différentes.

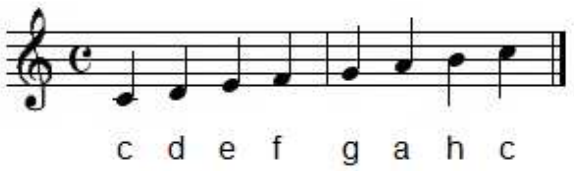
GAMMES

La gamme tempérée est basée sur une fréquence de référence de 440 Hz et divise l'octave (rapport de fréquence 2) en 12 demi-tons possédant tous le même rapport de fréquences *r*.

Cela donne donc:

$$r^{12} = 2 \quad \text{oder} \quad r = \sqrt[12]{2} = 1.0594630943$$

On peut ainsi générer les notes de la gamme de Do majeur formée de tons et demi-tons répartis de manière appropriée. La fréquence de référence correspond à la note La.



Dans la gamme naturelle, on obtient la fréquence des différentes notes par multiplication par le rapport *r* en partant de la fréquence de référence. Les rapports pour les 8 notes naturelles d'une octave sont donc:

$$1, \frac{9}{8}, \frac{5}{4}, \frac{4}{3}, \frac{3}{2}, \frac{5}{3}, \frac{15}{8}, 2$$

et correspondent aux rapports entretenus par les nombres entiers 24, 27, 30, 32, 36, 40, 45, 48. Pour jouer ces notes, il faut sauver ces fréquences dans une liste et les jouer à l'aide de la fonction *playTone()*. Une fois que la gamme tempérée et naturelle ont été jouées séparément, vous pouvez écouter les deux instruments accordés différemment jouer simultanément la gamme de Do majeur. Vous verrez que ça sonne horriblement mal.

```
r = 2**(1/12)
a = 440

scale_temp = [a / r**9, a / r**7, a / r**5, a / r**4, a / r**2,
              a, a * r**2, a * r**3]
scale_pure = [3/5 * a, 3/5 * a * 9/8, 3/5 * a * 5/4,  3/5 * a * 4/3,
              3/5 * a * 3/2, a, 3/5 * a * 15/8, 3/5 * a * 2]

playTone(scale_temp, 1000)
playTone(scale_pure, 1000)

playTone(scale_temp, 1000, block = False)
playTone(scale_pure, 1000)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Dans la gamme tempérée, les demi-tons successifs entretiennent tous le même rapport de fréquences, de sorte que la différence de fréquences n'est pas égale. L'avantage de la gamme tempérée par rapport aux gammes naturelles est que les rapports de fréquences restent les mêmes dans toutes les tonalités (Do majeur, Ré majeur, etc …) [**plus...**].

JOUER DES MÉLODIES

Il est également possible de jouer de simples mélodies pour le plaisir avec la fonction *playTone()*. Pour jouer des notes successives de même longueur, utilisons une liste de tuples indiquant chacun une fréquence et une durée. Il est également possible de choisir un instrument de musique. L'exemple suivant joue un air enfantin. Le reconnaîtrez-vous?

```
v = 250
playTone([("cdef", v), ("gg", 2*v), ("aaaa", v//2), ("g", 2*v),
          ("aaaa", v//2), ("g", 2*v), ("ffff", v), ("ee", 2*v),
          ("dddd", v), ("c", 2*v)], instrument="harp")
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

La facilité avec laquelle il est possible de jouer une mélodie avec *playTone()* est assez déconcertante. Par rapport à un instrument réel, le son est cependant relativement synthétique.

EXERCICES

1. On peut composer une mélodie en écrivant une liste de fréquences sonores et la rejouer ensuite grâce à une boucle *for* :

```
melody = [262, 444, 349, 349, 392, 330, 262, 466, 440, 392, 392, 349]
v = 200
for f in melody:
    playTone(f, v)
```

   a. Connaissez-vous cette chanson? Rejouer ce morceau un peu plus lentement

   b. Rejouer le morceau une octave plus haut.

   c. Le morceau est trop bas pour votre classe de chant et sa transposition à l'octave supérieure est trop haute. Transposez la mélodie de telle manière qu'elle démarre par un sol3 au lieu d'un do3.

2. Jouer l'accord de Do majeur composé des notes do4, mi4, sol4 (tierce, quinte) pendant 20 secondes avec la gamme tempérée. À cet effet, il est possible de passer à la fonction *playTone()* les lettres correspondant aux notes à jouer (Le do4 correspond à c'', le ré4 à d'' etc …). Rejouer le même accord en utilisant la gamme naturelle. Que remarquez-vous?

Documentation du système sonore

Sound

Fonction	Action
playTone(freq)	Joue un son de fréquence *freq* [Hz] et d'une durée de 1000 ms (fonction bloquante)
playTone(freq, block=False)	Idem, mais en version non bloquante. Utile pour jouer plusieurs sons simultanément
playTone(freq, duration)	Joue un son de fréquence *freq* sur une durée *duration* [ms]
playTone([f1, f2, ...])	Joue plusieurs sons successifs d'une durée de 1000 ms, selon les fréquences présentes dans la liste passée en paramètre
playTone([(f1, d1), (f2, d2), ...])	Idem, en spécifiant au sein d'un tuple, pour chaque fréquence de la liste, la durée pendant laquelle il faut tenir le son en [ms]
playTone([("c", 700), ("e", 1500), ...])	Idem, mais en spécifiant les fréquences à l'aide du nom des notes (**notation anglo-saxonne de Helmholz**) et les durées en [ms]. La tessiture supportée va de « C » qui correspond au Do1 jusqu'au « h''' » qui correspond au Si5 (voir l'article « **Fréquence des touches du piano sur Wikipedia** »)
playTone([("c", 700), ("e", 1500), ...], instrument = "piano")	Idem, en sélectionnant le type d'instrument. Les instruments supportés sont les suivants : piano, guitar, harp, trumpet, xylophone, organ, violin, panflute, bird, seashore, ... (voir la spécification MIDI)
playTone([("c", 700), ("e", 1500), ...], instrument = "piano", volume=10)	Idem, en spécifiant le volume compris entre 0 et 100

Importation du module : from soundsystem import *
Lecture

Fonction	Action
getWavMono(filename)	Retourne une liste d'échantillons correspondant au fichier mono *filename*. Avec "wav/xxx.wav", il est également possible de charger les fichiers depuis le sous-dossier _wav du dossier contenant l'archive *tigerjython2.jar*
getWavStereo(filename)	Idem, pour un fichier stéréo
getWavInfo(file)	Retourne une chaine de caractères contenant les informations à propos du fichier *file* telles que le taux d'échantillonnage, etc.
openSoundPlayer(filename)	Ouvre un lecteur sonore avec le fichier *filename* pour en permettre la lecture avec les fonctions de lecture présentées par la suite
openMonoPlayer(filename)	Idem, en ouvrant un lecteur mono. Il est possible d'ouvrir des fichiers stéréo en lecture mono en faisant la moyenne entre les deux canaux
openStereoPlayer(filename)	Idem, en ouvrant un lecteur stéréo. Il est possible d'ouvrir un fichier mono (les deux canaux seront alors identiques)
openSoundPlayerMP3(filename)	Comme *openSoundPlayer()*, mais pour des fichiers MP3
openMonoPlayerMP3(filename)	Comme *openMonoPlayer()*, mais pour des fichiers MP3
openStereoPlayerMP3(filename)	Comme *openStereoPlayer()*, mais pour des fichiers MP3
play()	Joue le son depuis la position actuelle et retourne immédiatement (fonction non bloquante)
blockingPlay()	Idem, mais en ne retournant que lorsque la lecture est terminée (fonction bloquante)

advanceFrames(n)	Avance la lecture de *n* échantillons depuis la position courante
advanceTime(t)	Idem, en avançant de *t* millisecondes depuis la position courante
getCurrentPos()	Retourne la position courante du curseur de lecture
getCurrentTime()	Retourne le temps de lecture courant
rewindFrames(n)	Recule le curseur de lecture de *n* échantillons à partir de la position de lecture courante
rewindTime(t)	Idem, en reculant de *t* millisecondes à partir de la position de lecture courante
stop()	Arrête la lecture et réinitialise le curseur de lecture à la position initiale
setVolume(v)	Ajuste le volume selon la valeur *v* comprise entre 0 et 100
isPlaying()	Retourne *True* si la lecture du clip n'est pas encore terminée
mute(bool)	Passe le système en mode muet lorsque le paramètre *bool* est *True* et audible lorsque *bool* est *False*.
playLoop()	Lecture en boucle : la lecture du clip est relancée indéfiniment depuis le début
replay()	Rejoue le clip une fois supplémentaire
delay(time)	Ajoute au programme un temps d'attente de *time* millisecondes, utile pour jouer des silences

Capture sonore et Enregistrement

openMonoRecorder()	Ouvre un enregistreur de sons mono
openStereoRecorder()	Ouvre un enregistreur de sons stéréo
capture()	Débute la capture sonore
stopCapture()	Suspend la capture sonore
getCapturedBytes()	Retourne les enregistrements capturés octet à octet sous forme de liste
getCapturedSound()	Retourne les échantillons enregistrés comme une liste de nombres entiers. En mode stéréo, la suite d'échantillons alterne entre chacun des deux canaux (les positions paires correspondent au premier canal et les positions impaires au deuxième canal)
writeWavFile(samples, filename)	Sauve les échantillons présents dans la liste d'échantillons *samples* dans le fichier WAV *filename*

Transformation de Fourier rapide (FFT = Fast Fourier Transform)

fft(samples, n)	Transforme les *n* premières valeurs de la liste *samples*. Retourne une liste de n // 2 valeurs spectrales équidistantes (floats) comprises entre *0* et *fs/2* Hz avec un écart (résolution) de *fs/n*
sine(A, f, t)	Génère une onde sinusoïdale d'amplitude *A* et de fréquence *f* (phase 0) pour chaque valeur *t*
square(A, f, t)	Idem, mais en générant une onde carrée
sawtooth(A, f, t)	Idem, mais en générant une onde en dents de scie
triangle(A, f, t)	Idem, mais en générant une onde triangulaire
chirp(A, f, t)	Génère, pour chaque valeur de *t*, une onde sinusoïdale d'amplitude *A* et dont la fréquence augmente avec le temps (fréquence initiale *f*)

chapitre five

ROBOTIQUE

Objectifs d'apprentissage

- Être capable de décrire ce qu'est un robot et de connaitre quelques applications de la robotique.
- Comprendre la différence entre un robot autonome et un robot contrôlé à distance, comprendre l'utilité de simuler un robot.
- Être capable de contrôler un robot EV3 ou NXT à l'aide d'un programme *Python*.
- Être capable d'expliquer par quelques exemples en quoi consiste un robot capable d'apprentissage. Faire la différence entre le mode apprentissage et le mode exécution.
- Connaître les principes d'un système de régulation et énumérer quelques exemples de régulateurs.
- Être capable de lire les valeurs d'un capteur par polling ou par gestionnaire d'événements depuis un programme.

"Les robots vont-ils hériter de la terre ? Oui, mais ils seront nos enfants."

Mais:

"On n'a jamais conçu de machine consciente de ce qu'elle fait ; mais la plupart du temps, nous ne le sommes pas non plus."

Marvin Minsky, Chercheur en IA au MIT

5.1 MODE RÉEL ET MODE SIMULATION

INTRODUCTION

On considère généralement comme un robot toute machine contrôlée par ordinateur capable de réaliser une activité d'ordinaire réservée à l'être humain. Si la machine est également capable de percevoir son environnement à l'aide de caméras et de capteurs et si elle peut y réagir de manière appropriée par ses actuateurs (moteurs, synthèse vocale, leds, …), on parle de **système intelligent**. Si le comportement d'un tel système imite l'humain, on parle de robot **androïde**.

Un exemple typique d'un tel système est le robot du film *WALL-E,* doué de sa propre conscience, à tel point qu'il recherche des pièces de rechange pour lui-même et qu'il se passionne pour des objets qu'il collectionne. Il est également capable de résoudre un Rubik's cube, ce qui est sans conteste généralement perçu comme un signe d'intelligence.

L'intelligence artificielle (IA), en anglais *Artificial Intelligence* (*AI*), traite de la question fondamentale consistant à décrire de manière précise ce qui caractérise un système intelligent. Pour répondre à cette question, il est nécessaire de définir au préalable ce que l'on entend par machine « intelligente ». Une approche possible à cette problématique reside dans le **test de Turing**.

Dans ce chapitre, nous nous pencherons sur des questions plus terre-à-terre et verrons comment gérer un robot équipé de capteurs tactile, photosensible, infrarouge et ultrasonique ainsi que de roues actionnées par deux moteurs électriques capables de faire avancer, reculer et même tourner le robot.

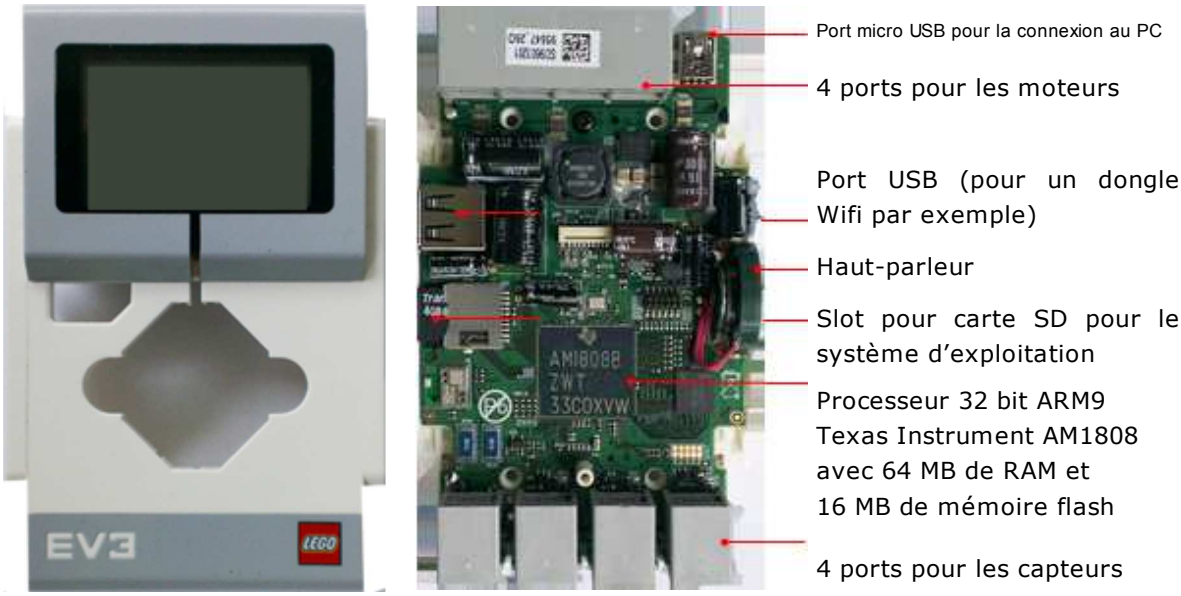
Les moteurs et les capteurs sont contrôlés par l'ordinateur de bord, raison pour laquelle un robot est parfois qualifié de **système embarqué**. S'il s'agit d'une simple puce électronique, on parle de microprocesseur ou microcontrôleur. De nos jours, les systèmes embarqués jouent un rôle très important et la plupart des appareils électroniques courants comme les smartphones en sont dotés. Il faut savoir que la plupart des machines à café, lave-linges, lave-vaisselles, télévisions, appareils photo et autres appareils électroniques sont des systèmes embarqués. Les voitures modernes sont équipées en moyenne d'une centaine de microcontrôleurs différents jouant le rôle de systèmes embarqués permettant de contrôleur le moteur, les gaz, le système ABS et autres. Sachez donc qu'en apprenant à comprendre le fonctionnement des robots, vous apprendrez les principes généraux qui régissent tous ces systèmes embarqués dont vous dépendez quotidiennement.

Si le processeur embarqué exécute un programme qui se suffit à lui-même, on parle d'un robot autonome. Le processeur embarqué, au lieu de contrôler lui-même le robot, peut également envoyer les valeurs lues sur les capteurs par une liaison de données quelconque vers un ordinateur externe qui se chargera alors d'effectuer le traitement et de lui renvoyer des instructions à exécuter en fonction des valeurs lues. On parle alors d'un **robot contrôlé à distance**. Finalement, un robot peut également être **simulé**, ce qui signifie qu'en lieu et place de

capteurs et moteurs matériels, on utilise des composants logiciels. Dans cette perspective, on fait correspondre à l'assemblage de composants physiques du monde réel un assemblage d'objets et de classes dans le monde simulé. En pratique, on commence généralement par simuler les robots sur ordinateur pour faciliter leur programmation et l'étude de leur comportement, ce qui permet d'éviter tout risque matériel ou environnemental.

Avec le fameux kit éducatif de robotique **LEGO Mindstorms**, il est possible d'étudier les aspects importants de la robotique en s'amusant. Le kit est constitué d'une **brique intelligente** contenant le microcontrôleur et plusieurs composants permettant l'élaboration de différents modèles de robots. Ce kit a subi plusieurs révisions : le premier de la série était le RCX, suivi par le NXT et, plus récemment, par le EV3.

La brique EV3 est un système embarqué constitué de moteurs et de capteurs pilotés par un processeur ARM semblable à ceux qui se trouvent dans les smartphones. Voici les composants électroniques observables si l'on ouvre la brique intelligente.



Une fois la brique allumée, le micrologiciel (firmware) va démarrer sur le microcontrôleur (il s'agit d'un OS Linux complet sur le EV3) faisant apparaître un menu très simple sur l'écran. Cela permet déjà d'exécuter des programmes stockés dans la mémoire de la brique en mode autonome. Pour permettre un contrôle à distance, il faut, dans le cas du EV3, lancer un programme annexe, BrickGate, chargé d'interpréter les commandes reçues par Bluetooth demandant par exemple de tourner un moteur d'un certain angle ou de lire les valeurs mesurées par l'un des capteurs. Sur le NXT, ce programme est déjà inclus dans le firmware.

Comme toujours en développement de systèmes embarqués, il faut un ordinateur externe pour développer les programmes. Ceux-ci sont téléchargés sur la brique en mode autonome et exécutés directement sur le PC en mode de contrôle à distance.

Mode autonome

Mode de contrôle à distance

CONCEPTS DE PROGRAMMATION: *Robots, robots androïdes, intelligence artificielle, systèmes embarqués, microprocesseurs, microcontrôleurs, méthodes bloquantes / non bloquantes*

PRÉPARATIFS

Avec *TigerJython*, il est possible de simuler le robot (**mode simulation**) ou d'utiliser le mode autonome ou de contrôle à distance **(mode réel)**. On passe d'un mode à l'autre en utilisant différentes bibliothèques de classes Python qui présentent cependant toutes exactement la même interface de programmation (API = Application Programming Interface) de sorte que les programmes seront pratiquement identiques quel que soit le mode d'exécution utilisé. Les seuls ajustement à apporter résident dans la ligne d'importation de modules et éventuellement dans certaines valeurs temporelles utilisées dans le programme.

Mode simulation :
Si vous ne disposez pas d'un kit NXT ou EV3, les activités de ce chapitre sont néanmoins réalisables en mode simulation. Les images nécessaires pour le mode simulation sont déjà incluses dans la distribution de *TigerJython*.

Mode réel :

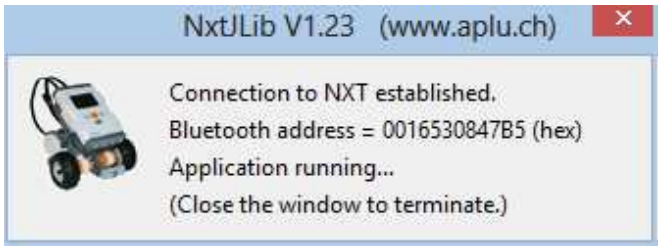
La plupart des exemples utiliseront le modèle de robot de base fourni avec le kit LEGO Mindstorms NXT ou EV3 qui est une plateforme à deux roues sur laquelle on peut disposer divers capteurs. Il est également possible d'utiliser un robot personnalisé, pourvu qu'il se déplace de manière identique. Du fait que le robot communique avec le PC par une liaison Bluetooth, il faut disposer d'un PC équipé d'un contrôleur Bluetooth activé. De plus, il est nécessaire de jumeler la brique intelligente au PC.

Avec la brique LEGO NXT:
Il est possible d'utiliser le firmware original du robot NXT ou le firmware alternatif LeJOS. Normalement, le nom du périphérique Bluetooth associé à la brique est « NXT ».

Pour jumeler la brique au PC, on procède exactement de la même manière que pour tout périphérique Bluetooth tel que smartphone, casque d'écoute ou imprimante Bluetooth.

Comme il n'est pas possible d'exécuter l'interpréteur Python directement sur le microcontrôleur du NXT en raison de ses ressources trop limitées, il faut obligatoirement utiliser le mode de contrôle à distance pour piloter le NXT à l'aide de Python. Pour ce faire, on développe un programme comme d'habitude dans *TigerJython* en important le module *ch.aplu.nxt* et en l'exécutant à l'aide du bouton vert « Exécuter ». Une boîte de dialogue demande alors le nom

Bluetooth de la brique à laquelle la connexion doit être établie. Une fenêtre affichant les informations de connexion reste ouverte durant toute l'exécution du programme. La fermeture de cette fenêtre cause l'interruption de la communication entre le PC et la brique NXT.

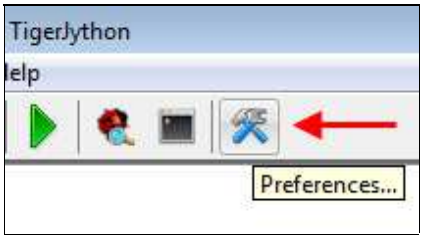


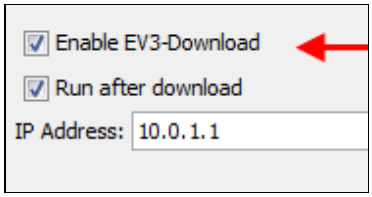
Si la salle contient plusieurs briques NXT, il est indispensable de leur attribuer des noms uniques tels que NXT1, NXT2, etc … Un outil permettant de changer le nom des briques est disponible **ici**. Il est également possible d'utiliser l'identifiant unique Bluetooth du périphérique au lieu d'utiliser son nom. L'identifiant est indiqué dans la fenêtre de connexion montrée ci-dessus et peut également se découvrir avec divers outils à disposition sur le Web. La bibliothèque BlueCove est nécessaire pour assurer la communication Bluetooth. Pour l'installer, il faut télécharger l'archive compressée disponible **ici** et la décompresser dans le sous-dossier *Lib* du dossier dans lequel l'archive *tigerjython2.jar* est située.

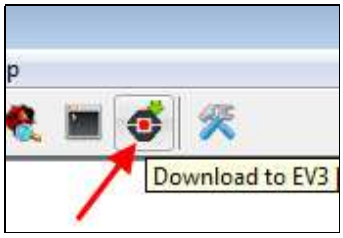
Avec la brique LEGO EV3:

La brique EV3 est nettement plus puissante que le NXT et exécute un système d'exploitation Linux conjointement avec LeJOS depuis la carte SD. Vous trouverez un guide détaillé pour créer la carte SD appropriée **ici**. En retirant la carte SD, il est possible d'utiliser le EV3 dans son état original. Si le EV3 est démarré en utilisant LeJOS, la communication s'opère par une liaison Bluetooth PAN. Pour ce faire, il faut appairer la brique au PC et la définir comme un point d'accès réseau. Vous trouverez des instructions détaillées **ici**.

Une fois la brique démarrée avec LeJOS et correctement connectée par un réseau PAN Bluetooth, il faut y démarrer le serveur ***BrickGate*** qui se trouve dans le menu « *programs* ».
Comme la brique EV3 est assez puissante pour faire tourner l'interpréteur Python, il est possible de l'utiliser en mode autonome avec des programmes Python, contrairement au NXT. Pour ce faire, il faut se rendre dans les préférences de *TigerJython*, dans le panneau *Librairie*, et cocher les deux options montrées dans les illustrations ci-dessous, à savoir « Sélection du robot = EV3 » et « Exécuter après téléchargement ». Ceci va faire apparaître un nouveau bouton « Télécharger vers EV3 » dans la barre d'outils de *TigerJython*.





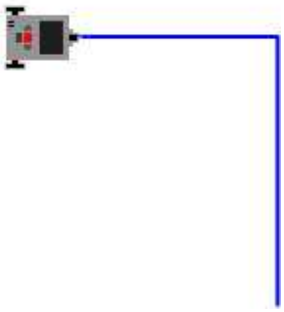


Pour exécuter le programme en contrôle à distance, il faut cliquer sur le bouton vert comme pour le NXT et entrer les informations de connexion à la brique par Bluetooth. Pour exécuter le programme en mode autonome directement sur la brique, il faut cliquer sur le bouton « Télécharger vers EV3 ». Le script Python est alors téléchargé sur le EV3 et exécuté de manière autonome par l'interpréteur Python installé sur la brique. Le nom du script apparaît également sur l'écran du EV3 et peut être relancé à tout moment, sans nécessité de connexion avec le PC, en appuyant sur le bouton « Enter » de la brique.

Les programmes exemples montrés dans ce chapitre supposent l'utilisation d'une brique EV3 et des nouveaux capteurs et moteurs de la série EV3. Le capteur de couleur EV3 joue également le rôle de capteur photosensible. Les anciens moteurs et capteurs NXT sont cependant encore supportés par la brique EV3. Il faut juste préfixer tous les noms de classes par « Nxt », à savoir NxtMotor, NxtGear NxtTouchSensor, etc.

AVANCER ET RECULER EN LIGNE DROITE, TOURNER

Dans votre premier programme, le robot devra avancer pendant une durée définie codée en dur dans le programme. La bibliothèque permettant de contrôler le robot est orientée objets et représente la réalité par un modèle. Ainsi, alors que dans la réalité on utilise la brique intelligente LEGO pour construire le robot, on crée une instance de la classe *LegoRobot()* du point de vue logiciel avec la ligne **robot = LegoRobot()**. Ensuite, on va prendre deux moteurs et les coupler à des roues dans la réalité alors que dans le monde logiciel, on instancie la classe *Gear* pour modéliser l'essieu de deux roues avec **gear = Gear()**. Ensuite, on connecte les moteurs à la brique avec des fils électriques dans le monde réel alors que, dans le monde logiciel, on « connecte » le couple des roues au robot avec l'appel **addPart()**.



La commande *gear.forward()* va faire tourner le couple des deux moteurs avec le même vitesse angulaire ce qui va faire avancer le robot en ligne droite. **Cet état de mouvement va rester identique jusqu'à ce que le robot reçoive un ordre différent**. Cependant, l'appel de fonction retourne immédiatement et le programme se poursuit avec l'exécution de la prochaine instruction. On appelle ceci une **méthode non bloquante**. De ce fait, il faut s'assurer que le robot fasse quelque chose d'autre que d'avancer après un certain laps de temps en invoquant la méthode **Tools.delay()** qui permet ou d'arrêter le mouvement ou de le modifier en envoyant une autre commande.

Dès qu'une nouvelle commande est envoyée au robot, l'état d'exécution actuel est terminé et remplacé par un nouvel état. Lorsque l'on travaille en mode de contrôle à distance, il faut toujours appeler la méthode **exit()** en fin de programme car elle est responsable de couper tous les moteurs et d'interrompre la connexion Bluetooth afin que le prochain programme puisse s'exécuter sans encombre. De ce fait, si le programme s'interrompt de manière brutale suite à une erreur d'exécution, il n'aura pas le temps d'appeler la méthode *exit()* et il sera nécessaire d'éteindre et de redémarrer la brique.

```
from simrobot import *
#from nxtrobot import *
#from ev3robot import *

robot = LegoRobot()
gear = Gear()
robot.addPart(gear)

gear.forward()
Tools.delay(2000)
gear.left()
Tools.delay(545)
gear.forward();
Tools.delay(2000)
robot.exit()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Un essieu est constitué de deux moteurs synchronisés. Au lieu de les contrôler individuellement, il est possible d'utiliser certaines commandes qui les contrôlent en tant que paire solidaire.

Les bibliothèques de classes gérant le mode simulé et réel sont conçues de telle manière que les programmes soient pratiquement identiques dans les différents cas, à la ligne d'importation

près. Ainsi, il est possible de commencer par développer le programme en mode simulation pour mettre en place et tester son comportement et, avec quelques adaptations mineures, l'exécuter ensuite en mode réel sur le robot physique. La brique EV3 peut se programme en mode autonome ou en contrôle à distance mais dans les deux cas, le serveur BrickGate doit être démarré sur le EV3 car c'est lui qui reçoit et interprète les commandes transmises par le programme *Python*. Du fait que les erreurs ne sont pas signalées en mode autonome, il faut toujours commencer par contrôler la brique à distance en exécutant le programme sur le PC (bouton vert) avant de passer au mode autonome et exécuter le programme Python directement sur le EV3 (bouton EV3).

FAIRE BOUGER LE ROBOT AVEC DES MÉTHODES BLOQUANTES

Au lieu de faire avancer le robot en ligne droite avec la commande *forward()* et de dire ensuite au programme de se mettre en pause pendant 2000 ms avec *delay(2000)*, il est également possible d'utiliser la méthode bloquante **forward(2000)** qui fait également avancer le robot en ligne droite mais en ne retournant qu'après 2000 ms. Il existe également des variantes bloquantes pour les commandes **left()** et *right()*.

Les méthodes bloquantes permettent ainsi de simplifier légèrement le précédent programme.

```
from simrobot import *
#from nxtrobot import *
#from ev3robot import *

robot = LegoRobot()
gear = Gear()
robot.addPart(gear)
gear.forward(2000)
gear.left(545)
gear.forward(2000)
robot.exit()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Il faut faire la distinction entre les méthodes bloquantes et les méthodes non bloquantes. Les méthodes non bloquantes ne font que de changer l'état d'exécution du robot en retournant immédiatement à l'appelant. Autre contraire, dans le cas d'un appel bloquant, l'exécution du programme est suspendue pendant l'intervalle de temps spécifié, jusqu'à ce que l'appel bloquant soit complètement terminé et que le temps indiqué soit écoulé.

À première vue, il peut paraître plus facile d'utiliser systématiquement de méthodes bloquantes mais il faut savoir qu'elles comportent un inconvénient majeur. Pendant que le programme est bloqué en attendant la fin de l'exécution de la fonction bloquante, il lui est impossible de faire autre chose d'utile comme lire les valeurs d'un capteur !

Si le programme se plante durant l'exécution, il est possible de l'interrompre en fermant la fenêtre de connexion à la brique présente dans TigerJython sur le PC. En cas d'urgence durant une exécution en mode autonome, on peut interrompre le programme en appuyant simultanément sur les boutons BAS+ENTER du EV3.

EXERCICES

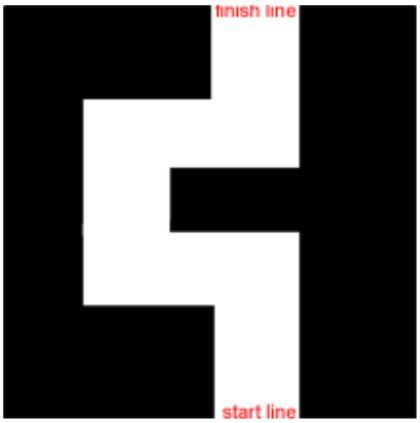
1. Développer un programme demandant au robot de se déplacer sur un carré en utilisant des méthodes bloquantes.

2. Écrire un programme utilisant de méthodes non bloquantes faisant en sorte que le robot se déplace sur des demi-cercles successifs.

3. Fabriquer un parcours avec quelques objets à portée de main et écrire un programme permettant au robot de se déplacer dans le parcours de puis le départ jusqu'à l'arrivée.

   Pour travailler en mode simulation, on peut utiliser l'image de fond *bg.gif* située dans le dossier *sprites* et l'afficher avec *RobotContext.useBackground()*.

   L'appel *RobotContext.setStartPosition()* permet alors de disposer le robot à une certaine position de départ au début du programme. Les coordonnées de la fenêtre s'étendent entre 0 et 500 dans les deux directions, l'origine se trouvant au coin supérieur gauche.




```
RobotContext.setStartPosition(200, 455)
RobotContext.useBackground("sprites/bg.gif")
```

   Vous pouvez également créer votre propre image de parcours qui doit avoir une taille de 501x501 pixels.

MATÉRIEL BONNUS

CONTRÔLE À DISTANCE PAR INFRAROUGE

Le EV3 est livré avec un capteur infrarouge assez polyvalent présentant de nombreux usages. Ce capteur est déjà livré dans le kit EV3 grand public mais doit être acheté séparément pour les kits éducatifs. Le tableau synoptique suivant montre les trois usages possibles du capteur IR (Infra Rouge).

Class	Grandeurs mesurées
IRSeekSensor	Distance et direction à la source IR de la télécommande
IRRemoteSensor	Boutons pressés de la télécommande
IRDistanceSensor	Distance à une cible réfléchissant les rayons IR

L'utilisation de la télécommande peut être amusante et motivante pour une première prise en main du robot. A contrario d'un programme de contrôle à distance prédéterminé, il est possible de définir, par simple programmation Python, les actions qui seront conduites par le robot en réponse aux différents boutons de la télécommande.

Dans le programme suivant, on décide d'utiliser de la manière suivante les différentes commandes possibles de la télécommande:

Boutons de la télécommande	**Action entreprise par le robot**
Haut-Gauche	Se déplacer sur un arc de cercle vers la gauche
Haut-Droite	Se déplacer sur un arc de cercle vers la droite
Left-Down+Right-Up	moves straight forward
Bas-Gauche	Arrête le robot
Bas-Droite	Termine le programme

```
from ev3robot import *

robot = LegoRobot()
gear = Gear()
robot.addPart(gear)
irs = IRRemoteSensor(SensorPort.S1)
robot.addPart(irs)
isRunning = True

while not robot.isEscapeHit() and isRunning:
    command = irs.getCommand()
    if command == 1:
        gear.leftArc(0.2)
    if command == 3:
        gear.rightArc(0.2)
    if command == 5:
        gear.forward()
    if command == 2:
        gear.stop()
    if command == 4:
        isRunning = False
robot.exit()
```

MEMENTO

Les méthodes *isEscapeHit(), isEnterHit(), isDownHit(), isUpHit(), isLeftHit(), isRightHit()* retournent *True* si les boutons correspondants de la brique EV3 sont actionnés en mode autonome. En mode contrôle à distance, ces fonctions sont par contre liées aux touches suivantes du clavier : ESCAPE, ENTER, BAS, HAUT, GAUCHE, DROITE à condition que la fenêtre de connexion à la brique soit active et possède le focus (il faut donc cliquer dessus pour que cela fonctionne).

5.2 ROBOTS INTELLIGENTS

INTRODUCTION

Les robots capables de trouver leur chemin dans un environnement dynamique présentent un potentiel d'applications important, notamment dans le domaine d'engins volants, dans l'exploration sous-marine ou dans l'examen des systèmes de canalisation d'égouts. Dans cette section, nous allons voir pas à pas comment construire un robot capable de s'orienter dans un environnement dynamique.

CONCEPTS DE PROGRAMMATION:

Robot auto-apprenant en mode autonome ou contrôlé à distance, mode apprentissage, mode exécution, boucle d'événements.

LE ROBOT CONNAÎT LE CHEMIN

Dans le cas le plus simple, un robot devrait être capable de trouver son chemin à l'intérieur d'un canal très spécial constitué d'éléments orthogonaux de même longueur.

L'information concernant la longueur constante des éléments du canal et si les virages tournent à gauche ou à droite est pour le moment codée en dur dans le programme.

```
from simrobot import *
#from nxtrobot import *
#from ev3robot import *

RobotContext.useObstacle("sprites/bg.gif", 250, 250)
RobotContext.setStartPosition(310, 470)

moveTime = 3200
turnTime = 545

robot = LegoRobot()
gear = Gear()
robot.addPart(gear)
gear.forward(moveTime)
gear.left(turnTime)
gear.forward(moveTime)
gear.right(turnTime)
gear.forward(moveTime)
gear.right(turnTime)
gear.forward(moveTime)
gear.left(turnTime)
gear.forward(moveTime)
```

```
robot.exit()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Il faut déterminer expérimentalement les bonnes valeurs pour **moveTime** et **turnTime** en réalisant plusieurs expériences successives et en ajustant les valeurs de manière appropriée. Il est clair que ces valeurs dépendent de la vitesse du robot. En situation réelle, on préfèrera spécifier le trajet à parcourir et les angles de rotation des roues plutôt que des durées pour les mouvements.

ROBOT CONTRÔLÉ PAR UN HUMAIN

Dans le programme suivant, le robot connaît la longueur constante des éléments du canal mais ses mouvements de rotation sont contrôlés par un humain. Il n'est pas encore capable d'effectuer un apprentissage pour se souvenir du chemin emprunté. Il demeure donc pour le moment « stupide ».

Afin de contrôler le robot en mode simulation ou en mode de contrôle à distance, il faut utiliser les touches directionnelles gauche et droite du clavier. Pour faire de même en mode autonome, il faudra utiliser les boutons gauche et droite de la brique EV3. Les méthodes *isLeftHit()* et *isRightHit()* permettent de déterminer si les touches directionnelles ou les boutons de la brique ont été pressés et relâchés. Pour terminer le programme, il faut utiliser la touche Escape du clavier.

```
from simrobot import *
#from nxtrobot import *
#from ev3robot import *

RobotContext.useObstacle("sprites/bg.gif", 250, 250)
RobotContext.setStartPosition(310, 470)

moveTime = 3200
turnTime = 545

robot = LegoRobot()
gear = Gear()
robot.addPart(gear)
gear.forward(moveTime)

while not robot.isEscapeHit():
    if robot.isLeftHit():
        gear.left(turnTime)
        gear.forward(moveTime)
    if robot.isRightHit():
        gear.right(turnTime)
        gear.forward(moveTime)
robot.exit()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Dans ce cas, il est moins intéressant d'utiliser le robot en mode autonome puisque l'on veut pouvoir le contrôler à distance à l'aide des touches du clavier. Il est également possible d'utiliser le capteur infrarouge et la télécommande au lieu du clavier pour contrôler le robot dans les virages (cf. le matériel supplémentaire disponible à la fin de cette section).

MODE APPRENTISSAGE

Des systèmes assistés par ordinateur dont le comportement n'est pas codé en dur dans le programme mais qui sont capables d'adapter leur comportement à leur environnement sont appelés **systèmes adaptatifs**. Ils sont donc plus ou moins capables d'effectuer des apprentissages. Les robots industriels sont « entrainés » par des spécialistes de la tâche à accomplir en « **mode apprentissage** ». Cela consiste par exemple à enseigner au robot les mouvements précis qu'il doit effectuer avec son bras robotisé. Dans la plupart des cas, l'opérateur du robot utilise un système d'entrée similaire à une télécommande. Le robot est ainsi déplacé successivement à la position désirée qui est immédiatement enregistrée.

En **mode exécution**, le robot parcourt les états mémorisés de manière indépendante en adoptant toutefois une vitesse bien supérieure.

Comme précédemment, notre robot de canalisation connaît la longueur exacte des divers éléments droits mais les virages sont encore contrôlés par un humain. Dans le programme suivant, le robot est cependant en mesure de retenir les virages que lui fait faire l'humain en mode apprentissage pour ensuite les reproduire à l'identique en mode exécution aussi souvent que nécessaire.

Il est souvent fort utile de s'imaginer que le robot se trouve à chaque instant dans un **état d'exécution** particulier. Ces états sont typiquement représentés par des mots significatifs et stockés sous forme de chaînes de caractères. Dans notre cas, on considère les états suivants du robot : être arrêté, avancer en ligne droite, tourner à gauche ou à droite. On peut nommer ces états STOPPED, FORWARD, LEFT, RIGHT [**plus...**]

Au lieu de vérifier constamment l'état des touches directionnelles du clavier ou des boutons de la brique (polling), il faut adopter une solution plus élégante utilisant le modèle de programmation événementielle. On y parvient en ayant recours à des fonctions de rappel qui sont d'abord enregistrées puis appelées automatiquement indépendamment du programme lorsque survient l'événement correspondant.

Le programme principal utilise une boucle infinie pour effectuer l'action correspondant à l'état courant du robot. Les changements d'état ont lieu dans la fonction de rappel *onButtonHit()*.

```
from simrobot import *
#from nxtrobot import *
#from ev3robot import *

RobotContext.useObstacle("sprites/bg.gif", 250, 250)
RobotContext.setStartPosition(310, 470)
RobotContext.showStatusBar(30)

def onButtonHit(buttonID):
    global state
```

```
    if buttonID == BrickButton.ID_LEFT:
        state = "LEFT"
    elif buttonID == BrickButton.ID_RIGHT:
        state = "RIGHT"
    elif buttonID == BrickButton.ID_ENTER:
        state = "RUN"

moveTime = 3200
turnTime = 545
memory = []
robot = LegoRobot(buttonHit = onButtonHit)
gear = Gear()
robot.addPart(gear)
state = "FORWARD"

while not robot.isEscapeHit():
    if state == "FORWARD":
        robot.drawString("Moving forward", 0, 3)
        gear.forward(moveTime)
        state = "STOPPED"
        robot.drawString("Teach me!", 0, 3)
    elif state == "LEFT":
        memory.append(0)
        robot.drawString("Saved: LEFT-TURN", 0, 3)
        gear.left(turnTime)
        state = "FORWARD"
    elif state == "RIGHT":
        memory.append(1)
        robot.drawString("Saved: RIGHT-TURN", 0, 3)
        gear.right(turnTime)
        state = "FORWARD"
    elif state == "RUN":
        robot.drawString("Executing memory", 0, 1)
        robot.drawString(str(memory), 0, 2)
        robot.reset()
        robot.drawString("Moving forward", 0, 3)
        gear.forward(moveTime)
        for k in memory:
            if k == 0:
                robot.drawString("Turning left", 0, 3)
                gear.left(turnTime)
            else:
                robot.drawString("Turning right", 0, 3)
                gear.right(turnTime)
            robot.drawString("Moving forward", 0, 3)
            gear.forward(moveTime)
        gear.stop()
        robot.drawString("All done", 0, 3)
        state = "STOPPED"
robot.exit()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

La lecture de l'état courant et sa traduction en une action sont effectuées dans une boucle *while* infinie au sein du programme principal et non dans les fonctions de rappel. En informatique, on appelle ce genre de boucles infinies une **boucle d'événements**. Les fonctions de rappel ne servent qu'à changer l'état du robot (state switch). Cette technique de programmation permet d'obtenir une synchronisation chronologique très claire entre les tâches de longue durée et les appels des fonctions de rappel en réponse aux événements qui peuvent survenir d'une seconde à l'autre. On utilise une liste pour mémoriser la succession des états appris en mode apprentissage en y stockant les nombres 0 ou 1 selon que le canal tourne à gauche ou à droite.

ROBOT CAPABLE D'APPRENTISSAGE NON SUPERVISÉ

Dans certaines situations, le robot doit être capable de s'adapter à son environnement et d'effectuer des apprentissages sans être supervisé par un opérateur humain. Le robot pourrait par exemple se trouver hors de portée de tout moyen de communication sur la planète Mars.

Pour retrouver son chemin, le robot doit pouvoir percevoir son environnement grâce à ses capteurs intégrés et réagir de manière appropriée. Les humains perçoivent essentiellement l'environnement par la vue. Il est facile pour les robots de capturer des images de leur environnement mais il leur est très difficile d'analyser et d'interpréter ces images [**plus...**].

Pour s'orienter dans la canalisation, notre robot n'utilisera qu'un capteur tactile qui déclenchera un événement lorsqu'il sera pressé. On suppose que les canalisations sont composées de segments rectilignes de longueur identique. Lorsque le robot détecte une pression sur le capteur tactile après avoir parcouru toute la longueur d'un segment, il sait qu'il se trouve face à un mur et qu'il doit tourner à gauche ou à droite. Il va donc reculer un peu pour se dégager du mur qui lui fait face et tente d'effectuer un virage à gauche. Si, presque immédiatement, il détecte à nouveau une interaction avec le capteur tactile, il sait qu'il a pris la mauvaise décision en tournant à gauche. Il va donc à nouveau reculer un petit peu et faire un demi-tour afin de partir vers la droite. Le robot va mémoriser le sens du virage à cet endroit du parcours de sorte qu'il pourra ensuite refaire le parcours sans se tromper dans les virages et en évitant, en théorie du moins, tous les murs.

Le programme suivant fait en sorte que le robot traverse le parcours en mode d'apprentissage non supervisé. Pour passer de la phase d'apprentissage au mode d'exécution, il faut presser la touche ENTER du clavier ou le bouton ENTER de la brique EV3.

```
from simrobot import *
#from nxtrobot import *
#from ev3robot import *

import time

RobotContext.useObstacle("sprites/bg.gif", 250, 250)
RobotContext.setStartPosition(310, 470)
RobotContext.showStatusBar(30)

def onPressed(port):
    global startTime
    global backTime
    robot.drawString("Press event!", 0, 1)
    dt = time.clock() - startTime # time since last hit in s
    gear.backward(backTime)
    if dt > 2:
        memory.append(0)
        gear.left(turnTime) # turning left
    else:
        memory.pop()
        memory.append(1)
        gear.right(2 * turnTime) # turning right
    robot.drawString("Mem: " + str(memory), 0, 1)
    gear.forward()
    startTime = time.clock()

def run():
    for k in memory:
        robot.drawString("Moving forward", 0, 1)
        gear.forward(moveTime)
        if k == 0:
            robot.drawString("Turning left", 0, 1)
            gear.left(turnTime)
        elif k == 1:
            robot.drawString("Turning right", 0, 1)
            gear.right(turnTime)
    gear.forward(moveTime)
```

```
    robot.drawString("All done", 0, 1)
    isExecuting = False

moveTime = 3200
turnTime = 545
backTime = 700
memory = []
robot = LegoRobot()
gear = Gear()

robot.addPart(gear)
ts = TouchSensor(SensorPort.S3, pressed = onPressed)
robot.addPart(ts)
startTime = time.clock()
gear.forward()
robot.drawString("Moving forward", 0, 1)

while not robot.isEscapeHit():
    if robot.isEnterHit():
        robot.reset()
        run()
robot.exit()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

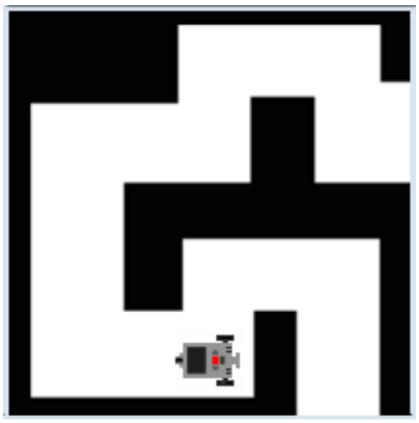
MEMENTO

Le capteur tactile est connecté au port S3, ce qui est interprété en mode simulation comme un positionnement du capteur en position centrale sur l'avant du robot. Le capteur signale les événements tactiles par l'intermédiaire de la fonction de rappel *onPressed()* qui est enregistrée avec le paramètre *pressed* du constructeur *TouchSensor()* . Le capteur tactile est un composant standard qui est rajouté au robot avec *addPart().* Pour déterminersi le robot a foncé dans une paroi en ligne droite ou, au contraire, après avoir fait une tentative infructueuse en tournant à gauche, on peut se baser sur le temps écoulé depuis le dernier événement généré par le capteur tactile. On utilise pour cela la fonction *clock()* du module *time* intégré à Python. Si ce temps est supérieur à deux secondes, c'est que le robot a simplement foncé dans le mur en ligne droite alors que, dans le cas contraire, il a fait une tentative de virage infructueuse et s'est retrouvé coincé. Du fait que le robot tourne toujours à gauche dans un premier temps en stockant un 0 dans sa mémoire, il doit remplacer ce 0 par un 1 s'il se retrouve bloqué suite à un faux virage.

UN ENVIRONNEMENT PLUS COMPLEXE

Vous avez probablement compris que le robot pourrait facilement déterminer par lui-même de combien il doit avancer en ligne droite sur chaque segment avant de tourner puisqu'il peut mesurer le temps qui s'écoule jusqu'à ce qu'il fonce dans un mur. De cette manière, le robot pourrait apprendre à se mouvoir dans un dédale de canalisation dont les segments sont de longueur variable. Cela nécessite cependant de mémoriser non seulement s'il doit tourner à gauche ou à droite, mais encore le temps de parcours de chacun des segments droits. Le plus simple est de stocker, pour chaque segment, ces deux informations dans une liste à deux éléments *node = [moveTime, k]*, où *moveTime* est le temps de parcours (en ms), *k = 0* correspond à un virage à gauche et *k = 1* à un virage à droite.

On obtiendra bien ainsi le temps *moveTime* nécessaire pour parcourir le segment mais il ne faudra pas oublier de le corriger en déduisant le temps pendant lequel le robot a foncé dans la paroi de la canalisation. Une fois ce temps corrigé, il faut le stocker dans une variable globale réutilisable dans la situation où le robot s'est retrouvé bloqué suite à un faux virage.



```
from simrobot import *
#from nxtrobot import *
#from ev3robot import *

import time

RobotContext.useObstacle("sprites/bg2.gif", 250, 250)
RobotContext.setStartPosition(410, 460)
RobotContext.showStatusBar(30)

def pressCallback(port):
    global startTime
    global backTime
    global turnTime
    global moveTime
    dt = time.clock() - startTime # time since last hit in s
    gear.backward(backTime)
    if dt > 2:
        moveTime = int(dt * 1000) - backTime  # save long-track time
        node = [moveTime, 0]
        memory.append(node) # save long-track time
        gear.left(turnTime) # turning left
    else:
        memory.pop() # discard node
        node = [moveTime, 1]
        memory.append(node)
        gear.right(2 * turnTime) # turning right
    robot.drawString("Memory: " + str(memory), 0, 1)
    gear.forward()
    startTime = time.clock()

def run():
    for node in memory:
        moveTime = node[0]
        k = node[1]
        robot.drawString("Moving forward",0, 1)
        gear.forward(moveTime)
        if k == 0:
            robot.drawString("Turning left",0, 1)
            gear.left(turnTime)
        elif k == 1:
            robot.drawString("Turning right",0, 1)
            gear.right(turnTime)
    gear.forward() # must stop manually
    robot.drawString("All done, press DOWN to stop", 0, 1)
    isExecuting = False

turnTime = 545
backTime = 700

robot = LegoRobot()
gear = Gear()
```

```
robot.addPart(gear)
ts = TouchSensor(SensorPort.S3, pressed = pressCallback)
robot.addPart(ts)
startTime = time.clock()
moveTime = 0
memory = []
gear.forward()

while not robot.isEscapeHit():
    if robot.isDownHit():
        gear.stop()
    elif robot.isEnterHit():
        robot.reset()
        run()
robot.exit()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

À première vue, la structure de données représentant la mémoire du robot en tant que liste de nœuds peut paraître compliquée. Il faut cependant toujours stocker des valeurs intrinsèquement liées dans une structure de données commune. En l'occurrence, il est indispensable d'associer la durée de parcours du prochain segment de canalisation et le sens dans lequel tourner à la fin dans une structure de données commune regroupant les deux informations. Admirez la manière dont le robot est capable, avec un seul et même programme, de retrouver son chemin dans un canal complètement différent (par exemple *bg3.gif*).

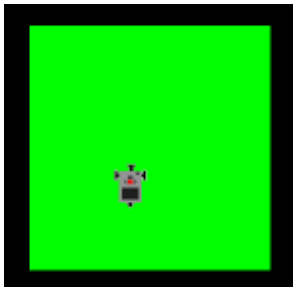
EXERCICES

1. Programmer le robot pour qu'il puisse trouver son chemin dans un canal comme l'image à droite. Le programme utilisera le capteur tactile pour détecter les parois sans pour autant effectuer d'apprentissage. Pour la phase de simulation, utiliser les options de *RobotContext* suivantes:

```
RobotContext.useObstacle("sprites/bg2gif", 250, 250)
RobotContext.setStartPosition(400, 470)
```

2. Programmer une tondeuse robotisée équipée d'un capteur tactile pour qu'elle soit capable de tondre le gazon par bandes verticales successives. La tondeuse fonce dans les murs supérieurs et inférieurs avant de tourner de manière appropriée.

   Pour la phase de simulation, utiliser les options de *RobotContext* suivantes:

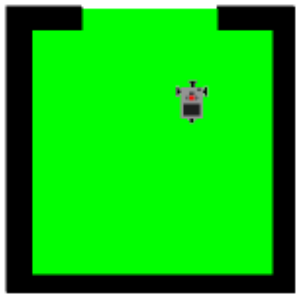


```
RobotContext.useBackground("sprites/fieldbg.gif")
RobotContext.useObstacle("sprites/field1.gif", 250, 250)
RobotContext.setStartPosition(350, 300)
```

3. Dans ce cas, la bordure supérieure du gazon est percée d'un trou par lequel la tondeuse robotisée peut s'échapper. Améliorer le programme de l'exercice précédent en rajoutant une composante d'apprentissage automatique permettant au robot de mémoriser la longueur des bandes après le premier passage afin d'éviter de devoir utiliser le capteur tactile pour détecter la bande supérieure.

Pour la phase de simulation, utiliser les options de *RobotContext* suivantes :

```
RobotContext.useBackground("sprites/fieldbg.gif")
RobotContext.useObstacle("sprites/field2.gif", 250, 250)
RobotContext.setStartPosition(350, 300)
```



MATÉRIEL SUPPLÉMENTAIRE

APPRENTISSAGE SUPERVISÉ À L'AIDE DE LA TÉLÉCOMMANDE IR

(uniquement en mode autonome sur le EV3)

Dans le dernier chapitre, vous avez déjà vu comment utiliser la télécommande IR pour commander le robot. Dans cette section, nous allons voir comment l'utiliser pour superviser l'apprentissage du robot. Le programme s'exécute en mode autonome directement sur le EV3, que ce soit en mode apprentissage ou en mode exécution.

Cette approche imite de manière réaliste la manière dont les robots industriels sont entraînés par une commande à distance pour mémoriser les actions à entreprendre qui sont simplement rejouées par la suite en mode exécution.

En mode apprentissage, l'opérateur utilise les deux boutons du haut de la télécommande pour faire tourner le robot à gauche ou à droite. Une pression sur le bouton inférieur gauche démarre le mode exécution. Une fois encore, il est élégant de travailler avec la notion d'états.

```
from ev3robot import *

def onActionPerformed(port, command):
    global state
    if command == 1:
        state = "LEFT"
    elif command == 3:
        state = "RIGHT"
    elif command == 2:
        state = "RUN"

moveTime = 3200
turnTime = 545
memory = []
robot = LegoRobot()
gear = Gear()
gear.setSpeed(50)
robot.addPart(gear)
irs = IRRemoteSensor(SensorPort.S1, actionPerformed = onActionPerformed)
robot.addPart(irs)
state = "FORWARD"
robot.drawString("Learning...", 0, 3)

while not robot.isEscapeHit():
    if state == "FORWARD":
        gear.forward(moveTime)
        state = "STOPPED"
    elif state == "LEFT":
        memory.append(0)
        gear.left(turnTime)
        gear.forward(moveTime)
        state = "STOPPED"
    elif state == "RIGHT":
```

```
        memory.append(1)
        gear.right(turnTime)
        gear.forward(moveTime)
        state = "STOPPED"
    elif state == "RUN":
        robot.drawString("Executing...", 0, 3)
        robot.reset()
        gear.forward(moveTime)
        for k in memory:
            if k == 0:
                gear.left(turnTime)
            else:
                gear.right(turnTime)
            gear.forward(moveTime)
        gear.stop()
        robot.drawString("All done", 0, 3)
        state = "STOPPED"
robot.exit()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

5.3 CONTRÔLE ET RÉGULATION

INTRODUCTION

Lorsque l'on travaille avec des machines, en particulier des robots, on est souvent confronté au problème de les contrôler de sorte qu'une variable mesurée soit aussi proche que possible d'une valeur prédéterminée, la **valeur de consigne**. C'est le cas dans une voiture équipée d'un tempomat responsable de maintenir la vitesse du véhicule aussi proche que possible d'une vitesse donnée, la valeur de consigne, indépendamment des facteurs extérieurs comme la pente de la route. À cette fin, un système de régulation doit être en mesure de déterminer la vitesse instantanée de la voiture (**valeur réelle**) à l'aide d'un capteur et d'ajuster ensuite la puissance du moteur de manière appropriée, comme si l'on agissait sur la pédale des gaz.
Voici quelques autres exemples de systèmes de régulation:

- Maintenir la température d'un réfrigérateur (contrôle par thermostat)
- Maintenir la trajectoire d'un avion sur un parcours déterminé (pilote automatique)
- Maintenir le niveau de remplissage d'un réservoir (par exemple la chasse d'eau des WC)

De nombreuses activités humaines peuvent être considérées comme des processus de régulation. Voici quelques exemples :

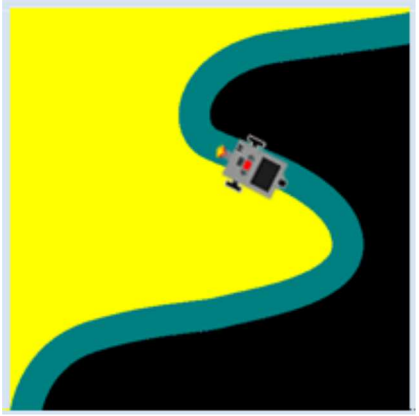
- Tourner le volant pour que la voiture reste sur la route
- Travailler juste ce qu'il faut pour obtenir son baccalauréat …
- Garder l'équilibre en se tenant sur une seule jambe

CONCEPTS DE PROGRAMMATION: *Système de régulation, valeur mesurée, valeur de consigne, incertitude de mesure*

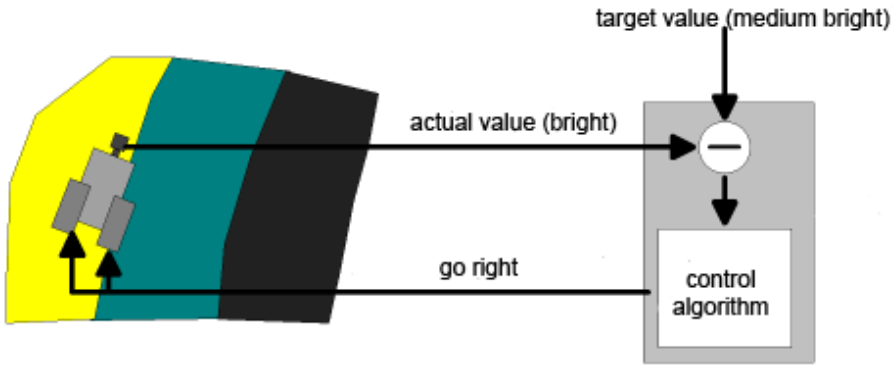
VOITURE AUTONOME

Conduire une voiture est un processus de contrôle très complexe devant prendre en compte de nombreux signaux d'entrée qui affectent le conducteur non seulement visuellement, mais de manière haptique (forces ressenties par le corps). Le comportement du conducteur résulte du traitement mental de tous ces signaux (tourner le volant, actionner la pédale des gaz, etc …).
Dans le future, les véhicules seront capables de conduire sans intervention humaine même dans des situations de trafique très dense. Plusieurs groupes de recherche de par le monde travaillent actuellement à ce problème et il n'est pas impossible que vous participiez vous-même un jour à ces recherches passionnantes. Dans cette section, vous pourrez déjà aiguiser vos compétences dans une situation extrêmement simplifiée.

Votre tâche est de guider le robot constitué d'un châssis monté sur roues et d'un capteur photosensible le long d'une route verte bordée d'une zone noire d'un côté et d'une zone verte de l'autre. Le capteur fonctionne en mesurant l'intensité de la lumière diffusée par le revêtement du sol. Le capteur mesure une intensité moyenne lorsqu'il se trouve sur la zone verte, élevée au-dessus de la zone jaune et faible au-dessus de la zone noire. Il incombe au système de régulation d'adapter l'intensité des moteurs en fonction de la valeur mesurée par le capteur photosensible de sorte que le robot soit capable de se déplacer le long de la route au mieux possible.



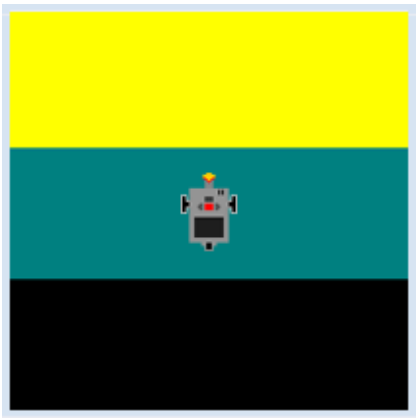
Schématiquement, on peut représenter ce processus par une **boucle de régulation**. Le capteur photosensible mesure l'intensité lumineuse (valeur mesurée) et la transmet au régulateur qui la compare à la valeur désirée (valeur de consigne) correspondant à la route verte. Le régulateur utilise la différence entre la valeur mesurée et la valeur de consigne pour déterminer par un algorithme de régulation spécifique à la situation la grandeur de contrôle à envoyer à l'essieu afin de ramener la valeur mesurée à la valeur de consigne.



Boucle de régulation

On voit que ce schéma constitue une boucle partant des capteurs du véhicule, passant par le système de régulation et revenant à nouveau vers les moteurs du véhicule qui vont à nouveau influencer les valeurs mesurées par les capteurs …

Avant de pouvoir coder le programme, il est nécessaire de connaitre les valeurs de référence mesurées par le capteur sur les différentes zones jaune, verte et noire. Pour déterminer ces valeurs, il suffit d'écrire un petit programme test qui affiche les valeurs mesurées par le capteur sur la console de TigerJython ou sur l'écran de la brique. En mode réel, il n'est pas nécessaire de programmer des déplacements car il suffit de prendre le robot dans les mains et de le placer sur la zone souhaitée. En mode simulation, il faut par contre déplacer le robot à travers les différentes zones colorées pour pouvoir observer les différentes valeurs mesurées par le capteur photosensible virtuel. On appelle ce processus la **calibration**. Il faut utiliser le capteur photosensible sur le NXT et le capteur colorimétrique sur la brique EV3.



```
from simrobot import *
#from nxtrobot import *
#from ev3robot import *
```

```
RobotContext.setStartPosition(250, 490)
RobotContext.useBackground("sprites/roadtest.gif")

robot = LegoRobot()
gear = Gear()
robot.addPart(gear)
ls = LightSensor(SensorPort.S3)
robot.addPart(ls)
ls.activate(True)
gear.forward()

while not robot.isEscapeHit():
    v = ls.getValue()
    print v
    Tools.delay(100)
robot.exit()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

Ce programme utilise un algorithme de régulation trivial : si la valeur mesurée est plus grande que la valeur de consigne, le véhicule se trouve dans la zone jaune et doit tourner vers la droite. Si la valeur mesurée est par contre inférieure à la valeur de consigne, le véhicule se trouve dans la zone noire et doit effectuer un virage vers la gauche. Dans tous les autres cas, il se trouve dans la zone verte et peut continuer sa course en ligne droite.

```
from simrobot import *
#from nxtrobot import *
#from ev3robot import *

RobotContext.setStartPosition(50, 490)
RobotContext.useBackground("sprites/road.gif")

robot = LegoRobot()
gear = Gear()
robot.addPart(gear)
ls = LightSensor(SensorPort.S3)
robot.addPart(ls)
ls.activate(True)
gear.forward()
nominal = 501

while not robot.isEscapeHit():
    actual = ls.getValue()
    if actual == nominal:
        gear.forward()
    elif actual < nominal:
        gear.leftArc(0.1)
    elif actual > nominal:
        gear.rightArc(0.1)

robot.exit()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

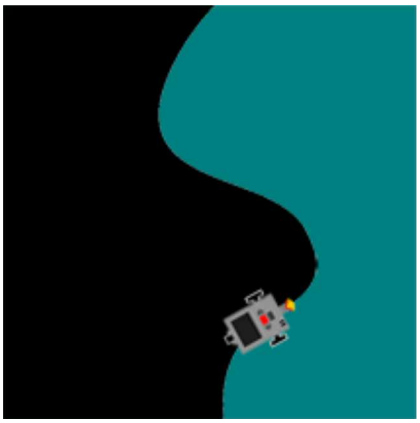
MEMENTO

La régulation fonctionne sans encombre en mode simulé, ce qui n'est pas le cas en mode réel. Ceci vient du fait que les valeurs mesurées par le capteur fluctuent dans la réalité, même lorsque le capteur se trouve sur une surface de couleur uniforme. Ces fluctuations sont dues, même sur une surface uniforme, aux différences d'éclairage et aux **erreurs de mesures** du capteur. Faites l'effort de chercher par vous-mêmes une solution à ce problème ! Le rayon de courbure transmis à *leftArc()* ou *rightArc()* est une paramètre sensible. Une petite valeur de ce paramètre conduit certes à de petits écarts avec la route mais à un comportement oscillatoire

très nerveux [**plus...**], tandis qu'une grande valeur du rayon de courbure permet un mouvement plus calme mais qui s'éloigne parfois excessivement de la route. Faites quelques expériences pour confirmer ce comportement en utilisant différents rayons de courbure.

EXERCICES

1. Déplacer le robot le long d'une frontière noire/vert à l'aide d'un capteur en utilisant le *RobotContext* suivant en mode simulé :



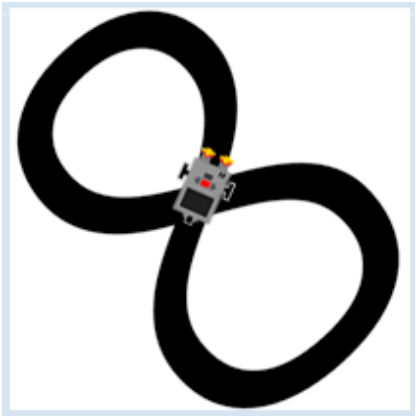
```
RobotContext.useBackground("sprites/edge.gif")
RobotContext.setStartPosition(250, 490)
```

2. Déplacer le robot le long d'un chemin elliptique en utilisant deux capteurs photosensibles. Utiliser le *RobotContext* suivant en mode simulé:

```
RobotContext.useBackground("sprites/roundpath.gif")
RobotContext.setStartPosition(250, 250)
RobotContext.setStartDirection(-90)
```

Régler la position et la direction de départ pour que le robot commence sa course sur la piste noire.

3. Parcourir un grand-huit à l'aide de deux capteurs photosensibles. Utiliser l'image de fond *track.gif* en mode simulation.



```
RobotContext.useBackground("sprites/track.gif")
```

5.4 TECHNOLOGIE DES CAPTEURS

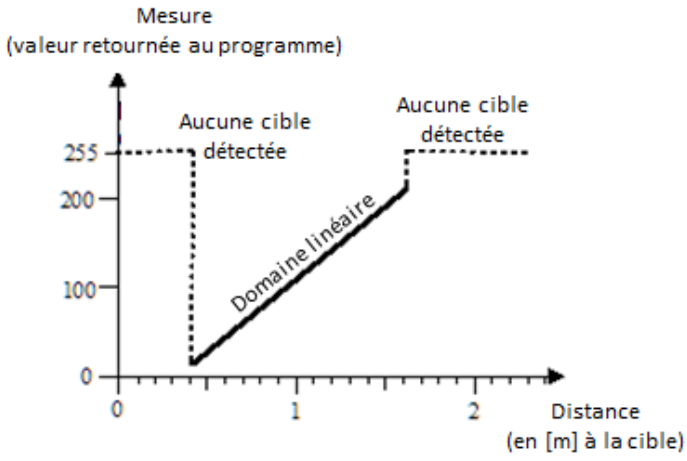
INTRODUCTION

Un capteur est un composant permettant de mesurer une grandeur physique telle que la température, l'intensité lumineuse, la pression ou des distances. Dans la plupart des cas, la valeur délivrée par le capteur est un nombre compris dans la plage des valeurs mesurées. Il existe cependant également des capteurs qui ne connaissent que deux états à la manière d'interrupteurs. On compte parmi ces derniers les capteurs tactiles, les capteurs permettant de savoir si le niveau d'un récipient est plein, etc.

La grandeur physique mesurée est généralement convertie par le capteur en une tension électrique qui est ensuite évaluée par de l'électronique supplémentaire. La structure interne d'un capteur peut être très complexe comme c'est le cas pour les capteurs ultrasoniques, gyroscopiques ou ceux permettant de mesurer des distances au LASER. La **courbe caractéristique** d'un capteur décrit la relation entre la grandeur physique mesurée et les valeurs délivrées par le capteur. De nombreux capteurs ont une courbe caractéristique plus ou moins linéaire mais il est toujours nécessaire de déterminer le facteur de conversion (pente) et l'ordonnée à l'origine. Pour ce faire, on procède au **calibrage** du capteur grâce à une série de mesures de grandeurs connues.

Le capteur ultrasonique permet de déterminer la distance à un objet en mesurant le temps nécessaire à une courte impulsion ultrasonique de voyager jusqu'à l'objet et d'être réfléchie jusqu'au capteur. Pour des distances comprises entre 30 cm et 2m, le capteur délivre des valeurs comprises entre 0 et 255 où 255 (-1 en mode simulation) est retourné lorsqu'il n'y a pas d'objet détecté à portée du capteur.

Capteur ultrasonique



Courbe caractéristique

Dans la plupart des applications, le programme interroge les valeurs lues par le capteur à intervalles réguliers. On appelle ceci « faire du polling ». Les valeurs d'un capteur sont donc lues et traitées en boucle. La résolution temporelle, qui correspond au nombre de mesures par seconde, dépend du capteur, de la rapidité de l'ordinateur et de la connexion à disposition pour relier la brique et le programme. Le capteur ultrasonique n'est par exemple capable de faire que deux mesures par seconde.

La valeur d'un capteur ne prenant que deux valeurs distinctes peut également être déterminée par polling. Il est cependant plus commode de concevoir les changements d'états comme des événements à traiter par programme à l'aide de fonctions de rappel déclenchées à chaque nouvelle lecture de valeur.

CONCEPTS DE PROGRAMMATION: *Capteurs, calibration de capteurs, polling et événements, seuil de déclenchement*

POLLING VS ÉVÉNEMENTS

Dans de nombreuses situations, il est possible de décider si l'on préfère contrôler un capteur par polling ou par gestionnaire d'événements. Tout dépend de l'utilisation que l'on veut en faire. On peut comparer les deux façons de procéder en connectant un moteur et un capteur tactile à la brique. Dans ce cas, une pression sur le bouton tactile devrait enclencher le moteur et une nouvelle pression devrait arrêter les moteurs.

Les événements sont bien plus malins pour cette application puisqu'ils informent de la pression sur le capteur tactile au travers d'un appel de fonction. Il suffit de passer cette fonction comme paramètre nommé lors de la création du capteur tactile avec *TouchSensor()*. Avec le polling, il est nécessaire d'utiliser un drapeau *isOff* (aussi appelé fanion, *flag* en anglais) dans le but de ne réagir que lors des transitions du capteur d'un état à l'autre.
En utilisant le polling et un fanion :

```
from nxtrobot import *
#from ev3robot import *

def switchMotorState():
    if motor.isMoving():
        motor.stop()
    else:
        motor.forward()

robot = LegoRobot()
motor = Motor(MotorPort.A)
robot.addPart(motor)
ts = TouchSensor(SensorPort.S3)
robot.addPart(ts)

isOff = True
while not robot.isEscapeHit():
    if ts.isPressed() and isOff:
        isOff = False
        switchMotorState()
    if not ts.isPressed() and not isOff:
        isOff = True
```

En utilisant les événements :

```
#from nxtrobot import *
from ev3robot import *

def onPressed(port):
    if motor.isMoving():
        motor.stop()
    else:
        motor.forward()

robot = LegoRobot()

motor = Motor(MotorPort.A)
robot.addPart(motor)
ts = TouchSensor(SensorPort.S1,
         pressed = onPressed)
robot.addPart(ts)
while not robot.isEscapeHit():
    pass
robot.exit()
```

MEMENTO

Les capteurs peuvent être gérés par polling ou par les événements. Il est indispensable de bien comprendre les deux techniques différentes et d'être en mesure de choisir de manière éclairée l'une ou l'autre selon la situation. Dans le modèle de la programmation événementielle, on définit des fonctions dont le nom commence conventionnellement par la chaine de caractères « on ». Ces dernières sont appelées **fonctions de rappel** puisqu'elles sont automatiquement appelées par le système en réponse aux événements survenus. Il faut **enregistrer** les fonctions de rappel en recourant à des paramètres nommés lors de la création de l'objet gérant le capteur.

INTERROGER UN CAPTEUR ULTRASONIQUE PAR POLLING

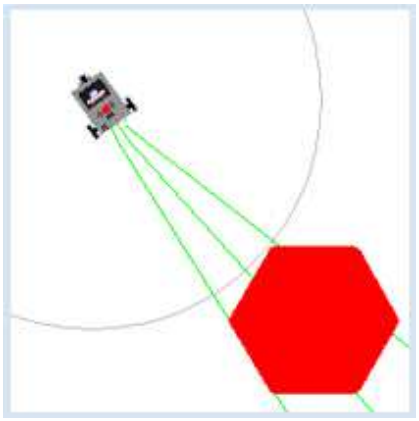
Note préliminaire : Si votre kit EV3 n'est pas livré avec un capteur ultrasonique, il est possible de réaliser cette section avec le capteur infrarouge.

Il est nécessaire d'interroger un capteur par polling si l'on a besoin d'un débit de mesures constant. Nous allons dans cette section permettre au robot de chercher un objet cible placé sur le sol quelque part dans la pièce, quelle que soit sa position de départ, pour le rejoindre.

Pour détecter une cible, on va recourir au capteur ultrasonique qui fonctionne exactement comme un système de radar. Pour apprendre à maîtriser un capteur, il n'y a rien de mieux que de développer un petit programme de test même si celui-ci n'est d'aucune utilité par la suite. Il est pratique et recommandé d'écrire les valeurs lues par le capteur dans la console Python et sur l'écran de la brique ainsi que de les traduire sous forme de son audible qui sera d'autant plus aigu que la distance sera grande. Ceci vous permettra de vous libérer les mains et les yeux tout en suivant en temps réel l'évolution des valeurs du capteur pendant les manipulations du robot à la main.

```
# from nxtrobot import *
from ev3robot import *

robot = LegoRobot()
us = UltrasonicSensor(SensorPort.S1)
robot.addPart(us)
isAutonomous = robot.isAutonomous()
while not robot.isEscapeHit():
    dist = us.getDistance()
    print "d = ", dist
    robot.drawString("d=" + str(dist), 0, 3)
    robot.playTone(10 * dist + 100, 50)
    if dist == 255:
        robot.playTone(10 * dist + 100, 50)
    if isAutonomous:
       Tools.delay(1000)
    else:
       Tools.delay(200)
robot.exit()
```

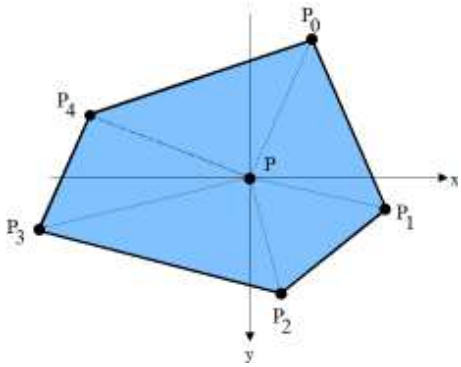
Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)



Pour trouver et s'approcher d'une cible, on peut tourner le robot comme une antenne de radar et chercher constamment la cible à l'aide du capteur ultrasonique. Lorsque la cible est détectée, il faut noter la direction actuelle et continuer de tourner jusqu'à ce que l'écho ne soit plus détecté. Ceci permet de détecter la taille apparente de l'objet, à savoir l'angle sous lequel l'objet est « visible ». On oriente ensuite le robot au milieu du cône de détection pour le diriger vers la cible et s'arrêter lorsqu'il est à la distance désirée de l'objet.

En mode simulation, on peut visualiser la distance mesurée à l'aide de **setBeamAreaColor()** et **setProximityCircleColor()**. La cible affichée correspond au fichier image spécifié comme paramètre à la fonction *RobotContext.useTarget()*.

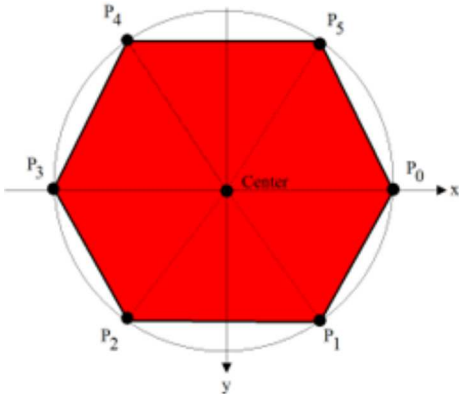
Cependant, la détection de la cible par le capteur virtuel ne se fera pas à l'aide de l'image affichée mais à l'aide d'un maillage (mesh en anglais) de triangles. Chacun de ces triangles est formé d'un point central *P* et de deux arrêtes. La cible affichée ci-contre est formée des meshes suivants :
PP0P1, PP1P2, PP2P3, PP3P4, PP4P0.



Dans le programme, on peut indiquer les sommets des triangles en tant que paramètre de la méthode *useTarget()*. Les coordonnées se réfèrent alors à un système de coordonnées dont l'origine se trouve au centre de la fenêtre, l'axe *Ox* étant dirigé vers la droite et l'axe *Oy* vers le bas.

Les coordonnées pour un hexagone de diamètre 100 sont par exemple

[50, 0] , [25, 43], [-25, 43], [-50, 0], [-25, -43], [25, -43].




```
from simrobot import *
#from nxtrobot import *
#from ev3robot import *

mesh = [[50, 0], [25, 43], [-25, 43], [-50, 0],
          [-25, -43], [25, -43]]
RobotContext.useTarget("sprites/redtarget.gif", mesh, 400, 400)

def searchTarget():
    global left, right
    found = False
    step = 0
    while not robot.isEscapeHit():
        gear.right(50)
        step = step + 1
        dist = us.getDistance()
        print "d = ", dist
        if dist != -1:  # simulation
        #if dist < 80:   # real
            if not found:
                found = True
                left = step
```

```
                print "Left at", left
                robot.playTone(880, 500)
        else:
            if found:
                right = step
                print "Right at ", right
                robot.playTone(440, 5000)
                break

left = 0
right = 0
robot = LegoRobot()
gear = Gear()
robot.addPart(gear)
us = UltrasonicSensor(SensorPort.S1)
robot.addPart(us)
us.setBeamAreaColor(makeColor("green"))
us.setProximityCircleColor(makeColor("lightgray"))
gear.setSpeed(5)

print "Searching..."
searchTarget()

gear.left((right - left) * 25)    # simulation
#gear.left((right - left) * 100)  # real

print "Moving forward..."
gear.forward()

while not robot.isEscapeHit() and gear.isMoving():
    dist = us.getDistance()
    print "d =", dist
    robot.playTone(10 * dist + 100, 100)
    if dist < 40:
        gear.stop()
print "All done"
robot.exit()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

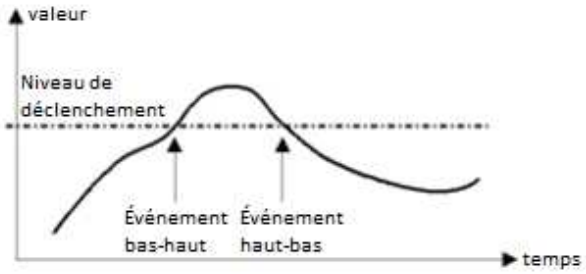
On peut généralement déterminer la valeur d'un capteur par des interrogations répétées à intervalles réguliers (polling) par l'entremise d'une méthode « getter » telle que *getValue()*, **getDistance()**, etc.

Lorsque l'on passe du mode simulation au mode réel, il est nécessaire d'ajuster certaines valeurs, en particulier les intervalles de temps. Il faut remarquer également que s'il ne détecte pas de cible, le capteur retourne -1 en mode simulation et 255 en mode réel.
En mode simulation, l'orientation du capteur ultrasonique par rapport au robot est déterminée par le port auquel il est connecté selon le tableau ci-dessous ::

Port capteur	Orientation du capteur
S1	Vers l'avant
S2	Vers la gauche
S3	Vers l'arrière

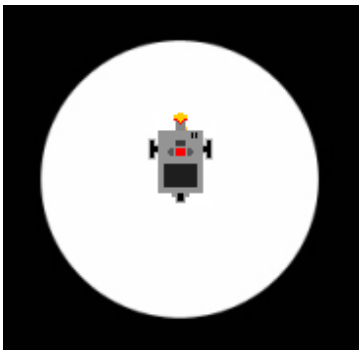
ÉVÉNEMENTS ET SEUILS DE DÉCLENCHEMENT

Les capteurs fournissant des données continues peuvent également être contrôlés par le modèle de programmation événementielle si l'on définit un **seuil de déclenchement** de l'événement. Le seuil de déclenchement correspondant à une valeur de la grandeur physique mesurée dont le franchissement dans un sens donné déclenche l'événement.



Les capteurs possèdent un seuil de déclenchement par défaut mais celui-ci peut être modifié par un appel à *setTriggerLevel()*.

Le programme suivant s'assure que le robot reste à l'intérieur d'une zone circulaire, pour éviter par exemple qu'il ne tombe d'une table. On utilise en l'occurrence le capteur photosensible de sorte qu'il ne réagisse qu'au contraste foncé / clair. Si la surface est foncée, le gestionnaire d'événements **onDark** est déclenché.



Lorsque l'on utilise le NXT en mode réel, il est très important d'allumer la LED rouge du capteur avec **activate(True)**.

```
from simrobot import *
#from nxtrobot import *
#from ev3robot import *

RobotContext.setStartPosition(250, 200)
RobotContext.setStartDirection(-90)
RobotContext.useBackground("sprites/circle.gif")

def onDark(port, level):
    gear.backward(1500)
    gear.left(545)
    gear.forward()

robot = LegoRobot()
gear = Gear()
robot.addPart(gear)
ls = LightSensor(SensorPort.S3,
      dark = onDark)
robot.addPart(ls)
ls.setTriggerLevel(100)  # adapt value
gear.forward()
while not robot.isEscapeHit():
    pass
robot.exit()
```

MEMENTO

Le franchissement d'un seuil par les valeurs mesurées par le capteur peut être interprété comme un événement. On appelle cela le **déclenchement** (triggering). Valeurs par défaut pour les seuils de déclenchement:

Capteur	**Niveau de déclenchement par défaut**
Capteur sonore	50
Capteur photosensible	500
Capteur ultrasonique	10

On peut résumer les avantages et les désavantages du modèle événementiel par rapport au polling de la manière suivante:

Avantages du modèle événementiel	**Désavantages du modèle événementiel**
Style de programmation plus clair et simplifié de fait que le code définissant le comportement est séparé du reste du programme dans une fonction de rappel (gestionnaire d'événement).	Le programme principal est interrompu de manière non prédictible (programmation asynchrone), ce qui peut interférer avec le flux d'instructions du programme principal.
L'événement est toujours détecté, même lorsque le PC est lent.	Les fonctions de rappel (gestionnaires d'événements) peuvent engendrer des effets de bord indésirables, en particulier lorsqu'elles modifient des variables globales ou l'état du robot.
Le programme peut continuer normalement et n'a pas à se préoccuper de capteurs.	Les fonctions de rappel sont exécutées dans un fil d'exécution séparé (thread), ce qui peut engendrer des conflits dus à la nature parallèle de l'exécution.
Le déclenchement est un concept central de la technique de mesure.	Une seule valeur de la grandeur mesurée peut être détectée, à savoir le seuil de déclenchement.
Le modèle événementiel correspond bien au modèle d'états d'exécution. L'événement change l'état du système.	Les fonctions de rappel ne devraient en principe pas contenir du code dont le temps d'exécution est très court pour éviter que les autres événements ne soient perdus.

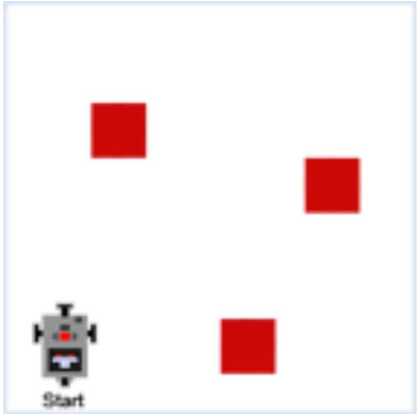
EXERCICES

1. Programmer le robot équipé du capteur sonore de telle manière qu'il démarre lorsqu'on frappe dans les mains pour la première fois. Lorsqu'on frappe des mains les fois suivantes, le robot doit changer de direction. Résoudre ce problème en mode simulation et en mode réel. En mode réel, il faut équiper le robot du capteur sonore. En mode simulation, il faut installer un microphone sur le PC et régler de manière appropriée le niveau d'enregistrement dans le panneau de configuration.

2. Connecter un moteur et un capteur tactile à la brique et développer un programme qui enclenche le moteur lors d'une pression sur le capteur tactile et qui l'éteint lors d'une deuxième pression.

3. Construire un robot équipé d'un capteur ultrasonique et d'un capteur tactile qui cherche trois objets fins (bougies, canettes, …), leur rentre dedans et les fait tomber.

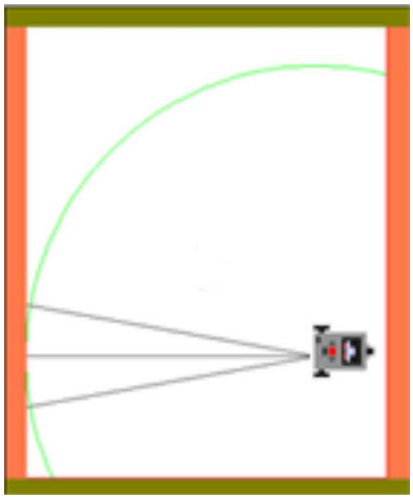
   En mode simulation, on peut interpréter la superposition avec une cible comme un événement tactile et utiliser *squaretarget.gif* (taille 60x60 pixels) pour représenter les objets à l'écran. On pourra utiliser le code suivant pour définir *RobotContext*. Essayez de bien comprendre les données dans la liste *mesh*.



```
mesh = [[-30, -30], [-30, 30], [30, -30], [30, 30]]
RobotContext.useTarget("sprites/squaretarget.gif", mesh, 350, 250)
RobotContext.useObstacle("sprites/squaretarget.gif", 350, 250)
RobotContext.useTarget("sprites/squaretarget.gif", mesh, 100, 150)
RobotContext.useObstacle("sprites/squaretarget.gif", 100, 150)
RobotContext.useTarget("sprites/squaretarget.gif" ,mesh, 200, 450)
RobotContext.useObstacle("sprites/squaretarget.gif", 200, 450)
RobotContext.setStartPosition(40, 450)
```

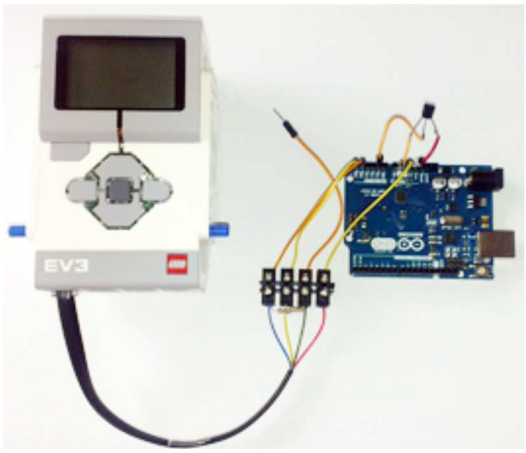
4*. Un robot équipé d'un capteur ultrasonique est placé au hasard dans une zone rectangulaire. Sa mission est de se placer aussi vite que possible aussi proche que possible du centre de la zone. Cette tâche peut être réalisée aussi bien en mode simulation qu'en mode réel.

En mode simulation, on peut utiliser les images *bar0.gif* et *bar1.gif* pour délimiter la zone rectangulaire à l'écran.



MATÉRIEL SUPPLÉMENTAIRE: CAPTEURS ARDUINO

Au contraire de la brique EV3, la carte de développement **Arduino** possède un système d'entrées/sorties standard avec des ports d'entrées digitaux. Elle dispose également de ports analogiques permettant de connecter des capteurs délivrant une tension proportionnelle à la grandeur mesurée. Cela permet l'utilisation d'une grande variété de capteurs et d'actuateurs bon marchés et d'y connecter des circuits électroniques bricolés. En connectant la carte Arduino à la brique EV3 au-travers d'une liaison appropriée, il est possible d'accéder à tous ces composants depuis un programme EV3. La connexion peut facilement se faire par une liaison I2C étant donné que les deux dispositifs supportent le protocole I2C.



En l'occurrence, la brique EV3 joue le rôle du maître I2C et l'Arduino celui de l'esclave. Les logiciels supplémentaires requis pour permettre cette liaison sont inclus d'office dans la distribution de TigerJython. La brique EV3 peut être opérée en mode direct (contrôle à distance) ou en mode autonome. Pour d'avantage d'informations, consulter la page **http://www.aplu.ch/ev3**.

Documentation des modules de robotique

Importations des module **from simrobot import ***
from ev3robot import *
from nxtrobot import *

LegoRobot:

Fonction	Action
LegoRobot()	Génère un objet *LegoRobot* modélisant une brique NXT, EV3 ou simulé dépourvue de moteur et de capteurs tout en établissant une connexion à la brique. Les capteurs et moteurs doivent ensuite être ajoutés avec *addPart()*
addPart(part)	Ajoute un composant (capteur, moteur ou châssis) au robot
clearDisplay()	Réinitialise l'écran [*mode simulation:* barre d'état]
drawString(text, x, y)	Écrit le texte *text* à la position *x, y* [*Mode simulation :* dans la barre d'état, (x, y) est dans ce cas ignoré]
isEnterHit()	Indique *True* si le bouton ESCAPE de la brique était pressé [*Sur le NXT et en mode simulation :* correspond à la touche *Esc*]
isLeftHit()	Indique *True* si le bouton LEFT de la brique était pressé [*Sur le NXT et en mode simulation :* correspond à la touche directionnelle *gauche*]
isRightHit()	Indique *True* si le bouton RIGHT de la brique était pressé [*Sur le NXT et en mode simulation :* correspond à la touche directionnelle *droite*]
isUpHit()	Indique *True* si le bouton UP de la brique était pressé [*Sur le NXT et en mode simulation :* correspond à la touche directionnelle *haut*]
playTone(frequency, duration)	Joue un son de fréquence *frequency* (en Hz) et de durée *duration* (ms) [*Simulation mode:* non disponible]
setVolume(volume)	Règle le volume pour la restitution des sons (Compris entre 0 et 100)
setLED(pattern)	Ajuste les LEDS du EV3: 0: éteint, 1: vert, 2: rouge, 3: rouge brillant, 4: vert clignotant, 5: rouge clignotant, 6: rouge brillant clignotant, 7: double clignotement vert, 8: double clignotement rouge, 9: double clignotement rouge brillant
exit()	Arrête le robot et termine la connexion
isConnected()	Indique *True* si la connexion est établie ou si la fenêtre de simulation n'est pas fermée
reset()	*En mode simulation :* place le robot dans la position / direction de départ

Gear (Essieu = deux moteurs synchronisés):

Gear()	Génère un objet *Gear* modélisant un châssis équipé d'un essieu constitué de deux moteurs synchronisés connectés sur les ports A et B
backward()	Fait rouler le châssis en arrière (méthode non bloquante)
backward(ms)	Roule en arrière pendant l'intervalle de temps *ms* (méthode bloquante)
isMoving()	Idem, en tournant pendant *ms* millisecondes (méthode bloquante)
forward()	Fait avancer le châssis (méthode non bloquante)
forward(ms)	Fait avancer le châssis pendant l'intervalle de temps indiqué par *ms* (méthode bloquante)
left()	Tourne à gauche (méthode non bloquante)

left(ms)	Tourne à gauche pendant *ms* millisecondes (méthode bloquante)
leftArc(radius)	Tourne à gauche en formant un arc de cercle de rayon radius (méthode non bloquante)
leftArc(radius, ms)	Idem, en tournant pendant *ms* millisecondes (méthode bloquante)
right()	Tourne à droite (méthode non bloquante)
right(ms)	Idem, en tournant pendant *ms* millisecondes (méthode bloquante)
rightArc(radius)	Tourne à droite en formant un arc de cercle de rayon radius (méthode non bloquante)
rightArc(radius, ms)	Idem, en tournant pendant *ms* millisecondes (méthode bloquante)
setSpeed(speed)	Règle la vitesse du robot
stop()	Arrête le châssis / l'essieu
getLeftMotorCount()	Retourne la valeur actuelle du compteur du moteur gauche [non en mode simulation]
getRightMotorCount()	Retourne la valeur actuelle du compteur du moteur droit [non disponible en mode simulation]
resetLeftMotorCount()	Remet à 0 le compteur du moteur gauche [non disponible en mode simulation]
resetRightMotorCount()	Remet à 0 le compteur du moteur droit [non disponible en mode simulation]

TurtleRobot:

TurtleRobot()	Génère un objet *TurtleRobot* modélisant un robot ayant un comportement de tortue. Le robot doit être équipé d'un châssis monté sur un essieu entraîné par les moteurs connectés aux ports A et B
backward()	Recule (méthode non bloquante)
backward(step)	Recule du nombre de pas spécifié (méthode bloquante)
forward()	Avance (méthode non bloquante)
forward(step)	Avance du nombre de pas spécifié (méthode bloquante)
left()	Tourne à gauche (méthode non bloquante)
left(angle)	Tourne à gauche (méthode non bloquante)
right()	Tourne à droite (méthode non bloquante)
right(angle)	Tourne à droite de l'angle *angle* (méthode bloquante)
setTurtleSpeed(speed)	Règle la vitesse du châssis à *speed*

Motor:

Motor(MotorPort.port)	Génère un objet *Motor* modélisant un moteur virtuel branché sur le port moteur A, B, C, ou D
backward()	Fait tourner les moteurs à l'envers
forward()	Fait tourner les moteurs à l'endroit
setSpeed(speed)	Règle la vitesse du moteur
isMoving()	Retourne *True* si le moteur est en rotation
stop()	Arrête le moteur
getMotorCount()	Retourne la valeur actuelle du compteur de moteur [non disponible en mode simulation]
resetMotorCount()	Réinitialise le compteur du moteur à 0 et le fait tourner jusqu'à ce que le compteur atteigne la valeur *count*. (méthode bloquante) [non disponible en mode simulation]

rotateTo(count)	Idem que *rotateTo(count)*, mais de manière non bloquante lorsque *blocking = False* [non disponible en mode simulation]
rotateTo(count, blocking)	Idem que *rotateTo(count)*, mais de manière non bloquante lorsque *blocking = False* [non disponible en mode simulation]
continueTo(count)	Idem que *rotateTo(count, False)*, mais sans réinitialiser le compteur à 0 [non disponible en mode simulation]
continueTo(count, blocking)	Idem que *rotateTo(count)*, mais de manière non bloquante lorsque *blocking = False* [non disponible en mode simulation]
continueRelativeTo(count)	Idem que *continueTo(count)*, mais avec un incrément de *count* pour le compteur [non disponible en mode simulation]
continueRelativeTo(count, blocking)	Idem que *continueTo(count, blocking)*, mais avec un incrément de *count* pour le compteur [non disponible en mode simulation]

LightSensor:

LichtSensor(SensorPort.port)	Génère un objet *LightSensor* modélisant un capteur optique (photosensible) branché sur le port S1, S2, S3, ou S4
LightSensor(SensorPort.port, dark = onDark)	Enregistre la fonction de rappel *onDark*
LightSensor(SensorPort.port, bright = onBright)	Enregistre la fonction de rappel *onBright*
activate(True)	Active la LED du capteur photosensible, ce qui est nécessaire pour mesurer l'intensité lumineuse diffusée par une surface au sol (uniquement sur la brique NXT)
activate(False)	Désactive laLED du capteur photosensible
getValue()	Indique la valeur lue par le capteur photosensible (nombre 0 .. 1000)
setTriggerLevel(level)	Ajuste le seuil de déclenchement

ColorSensor:

ColorSensor(SensorPort.port)	Génère un objet *ColorSensor* modélisant un capteur de couleurs branché sur un des ports S1, S2, S3, ou S4
getColor()	Retourne la couleur mesurée par le capteur sous forme d'un objet *Color* possédant les méthodes *getRed()*, *getGreen()* et *getBlue()* permettant d'obtenir les composantes RVB comprises entre 0 et 255.
getColorID()	Retourne l'identifiant de la couleur actuellement mesurée : 0: indefini, 1: noir, 2: bleu, 3:vert, 4: jaune, 5: rouge, 6: blanc
getColorStr()	Retourne la couleur mesurée par une chaine de caractères (BLACK, BLUE, GREEN, YELLOW, RED, WHITE ou UNDEFINED)
getLightValue()	Retourne la luminosité de la couleur mesurée selon le modèle TSL = Teinte, Saturation, Lumière, (HSG = Hue, Saturation, Lightness)

TouchSensor:

TouchSensor(SensorPort.port)	Génère un objet *TouchSensor* modélisant un capteur tactile branché sur un des ports S1, S2, S3, ou S4
TouchSensor(SensorPort.port, pressed = onPressed, released = onReleased)	Idem, en rattachant les gestionnaires d'événements *onPressed(port), onReleased(port)*
isPressed()	Retourne *True* si le capteur tactile est enfoncé

SoundSensor:

SoundSensor(SensorPort.port)	Génère un objet *SoundSensor* modélisant un capteur sonore branché sur un des ports S1, S2, S3, ou S4.Pour EV3 utiliser le capteur type NXT: NxtSoundSensor(SensorPort.port)
getValue()	Retourne le niveau sonore mesuré
setTriggerLevel(level)	Règle le seuil de déclenchement pour les événements

UltrasonicSensor:

UltrasonicSensor(SensorPort.port)	Génère un objet *UltrasonicSensor* modélisant un capteur ultrasonique branché sur un des ports S1, S2, S3, ou S4
getDistance()	Retourne la distance mesurée par le capteur en [cm]. Retourne 255 si la mesure échoue (aucune cible)
setTriggerLevel(level)	Règle le seuil de déclenchement (valeur par défaut : 10)
far(port, level), near(port, level)	Paramètres nommés du constructeur *UltrasonicSensor* permettant d'enregistrer un gestionnaire d'événements (fonction de rappel). La fonction de rappel *far = onFar* est déclenchée lorsque la distance passe au-dessus du seuil de déclenchement et *near = onNear* lorsqu'elle passe au-dessous du seuil
setProximityCircleColor(color)	En mode simulation : Règle la couleur du cercle de proximité
setMeshTriangleColor(color)	En mode simulation : Règle la couleur des triangles de maillage (mesh)
eraseBeamArea()	En mode simulation : Efface le cône de détection de l'écran

InfraredSensor (only EV3):

IRRemoteSensor(SensorPort.port)	Génère un objet *IRRemoteSensor* modélisant un capteur infrarouge branché sur un des ports S1, S2, S3, ou S4
getCommand()	Retourne l'ID de la commande courante : 0:Rien, 1:Supérieur-gauche, 2: Inférieur-droit, 3:Supérieur-droit, 4:inférieur-droit, 5:Supérieur-gauche&supérieur-droit, 6:Supérieur-gauche&inférieur-droit, 7:inférieur-gauche&supérieur-droit, 8:inférieur-gauche&inférieur-droit, 9:Bouton-central, 10:inférieur-gauche&supérieur-gauche, 11:Supérieur-droit&inférieur-droit. Le canal est sélectionné par le slider rouge. 1=position supérieure, 4= position inférieure. Cela correspond au numéro de port sur lequel est attaché le capteur
actionPerformed(port, command)	Paramètres nommés du constructeur *UltrasonicSensor* permettant d'enregistrer un gestionnaire d'événements *onActionPerformed* (fonction de rappel)
IRSeekSensor(SensorPort.port)	Génère un objet *IRSeekSensor* modélisant un capteur infrarouge de recherche branché sur un des ports S1, S2, S3, ou S4. La source IR active de la télécommande doit être activée (bouton central de la télécommande)
v = getValue()	*v.bearing* spécifie la direction (-12..12) et *v.distance* la distance (en cm) à la source IR. Le canal est sélectionné par le slider rouge. 1=position supérieure, 4= position inférieure. Cela correspond au numéro de port sur lequel est attaché le capteur
IRDistanceSensor(SensorPort.port)	Génère un objet *IRDistanceSensor* modélisant un capteur infrarouge de mesure de distances branché sur un des ports S1, S2, S3, ou S4. La cible doit être en mesure de réfléchir le rayonnement IR
getDistance()	Retourne la distance (en cm) à la cible

TemperatureSensor (only EV3)):

TemperatureSensor(SensorPort.port)	Génère un objet *TemperatureSensor* modélisant un capteur de température (thermomètre) branché sur un des ports S1, S2, S3, ou S4. Modèle de capteur NXT géré : Temperature Sensor 9749
getTemperature()	Retourne la température en degrés Celsius comprise -55 .. 128

ArduinoLink (only EV3):

ArduinoLink(SensorPort.port)	Crée un objet *ArduinoLink* modélisant un maître I2C permettant une connexion à une carte de développement Arduino sur un des ports S1, S2, S3, ou S4
getReply(request, reply)	Envoie la requête (nombre entier entre 0 et 255) vers l'Arduino et retourne la réponse en modifiant directement la liste *reply* comportant au maximum 16 nombres entiers compris entre 0 et 255
getReplyInt(request)	Idem, mais en fournissant la réponse sous la forme de la valeur de retour (nombre entier 0 et 255).
getReplyString(request)	Idem, mais en fournissant la réponse sous la forme de la valeur de retour (chaîne de caractères de longueur maximale 15).

I2CExpander (only EV3):

I2CExpander(SensorPort.port, deviceType, slaveAddress)	creates an I2C expander at SensorPort S1, S2, S3, S4. deviceType = 0: PCF8574, 1: PCF8574A, 2: PCF8591; slaveAddress: 8-bit I2C address
I2CExpander(SensorPort.port, deviceType, inputMode, slaveAddress)	same, but defines inputMode: 0: single ended, 2: three differential, 3: mixed, 4: two differential (see data sheet PCF8591)
writeDigital(out)	sets the digital input/output (8 bits) and returns the current value. To define a pin as input, the port bit is set to 1. (Only for PCF8574/PCF8574A)
writeAnalog(out)	sets the analog ouput (8 bits). (Only for PCF8591)
readAnalog(channel)	returns the current value of channel 0..3 (0..255 for single ended, -128..127 for differential). (Only for PCF8591)
readAnalog()	returns list of currenet values of all channels. (Only for PCF8591)

RobotContext (nur Simulation)

setStartDirection(angle)	Règle la position de départ du robot (0 = orienté vers l'Est, les angles sont orientés dans le sens des aiguilles de la montre)
setStartPosition(x, y)	Règle la position de départ du robot (en pixels, origine dans le coin supérieur gauche du canevas)
showStatusBar(height)	Ajoute une barre d'état de hauteur *height* au bas de la fenêtre
setStatusText(text)	Ajoute le texte *text* dans la barre d'état en effaçant le texte présent
useBackground(filename)	Insère l'image contenue dans le fichier *filename* au fond. Celle-ci peut être détectée par le capteur virtuel photosensible ou de couleur comme une surface de revêtement.
useObstacle(filename, x, y)	Insère un obstacle à la position *(x, y)* pouvant être détecté par le capteur virtuel tactile.
useTarget(filename, mesh, x, y)	Insère une cible à la position *(x, y)* pouvait être détectée par le capteur virtuel ultrasonique. La représentation à l'écran de l'obstacle est spécifiée par l'image contenue dans le fichier *filename* et le maillage triangulaire (mesh triangles) dans la liste de sommets *mesh.*

chapitre six

INTERNET

Objectifs d'apprentissage

- Connaître les chaînes de caractères en tant que type de données et être capable de travailler avec les méthodes importantes qu'elles mettent à disposition.
- Savoir ce qu'est un document HTML et connaître quelques balises HTML.
- Être capable d'ouvrir un fichier HTML depuis le système de fichier ou en le téléchargeant depuis un serveur Web pour l'afficher dans une fenêtre de navigateur.
- Comprendre le modèle client-serveur et être capable de demander une ressource Web à un serveur Web à l'aide d'une requête HTTP GET.
- Être capable d'analyser un document HTML par programme pour y chercher une information précise.
- Être capable de décrire le type de données *dictionnaire* et savoir dans quels cas il est particulièrement utile.
- Être capable d'effectuer une recherche Google par programme.

6.1 HTML, CHAÎNES DE CARACTÈRES

INTRODUCTION

HTML (Hyper Text Markup Language) est un langage de description de documents conçu pour les pages Web. Le rendu d'un site Web dans le navigateur, aussi complexe puisse-t-il paraître, est généré à partir d'un fichier texte ordinaire contenant un **balisage** (*markup* en anglais) marquant la structure du document en plus du texte visible. Ce balisage est constitué de paires de **balises** (*tags* en anglais) formées d'une balise ouvrante et d'une balise fermante. La balise ouvrante débute par un chevron < et se termine par son opposée > ; la balise fermante débute par </ et se termine par >.

La structure de base d'un fichier texte HTML est constituée des balises <*html*> et <*body*> et de leur balise fermante correspondante.

```
<html>
   <body>
      TigerJython Web-Site
   </body>
</html>
```

La casse des caractères formant les balises (majuscules ou minuscules) ainsi que les indentations et les retours à la ligne importent peu pour le rendu final du document. On aurait donc très bien pu obtenir le même résultat avec le fichier suivant qui n'est cependant pas recommandé puisqu'il est moins lisible et moins uniforme pour un humain.

CONCEPTS DE PROGRAMMATION: *HTML, lien hypertexte, chaine de caractères, type de donnée constant (non mutable)*

QU'EST-CE QU'UNE CHAÎNE DE CARACTÈRES?

Dans la plupart des programmes, particulièrement dans le contexte du Web, il faut un type de donnée capable de stocker du texte formé par un enchaînement de caractères individuels issu, par exemple, du clavier ou d'un fichier. Il est nécessaire de disposer également de certains caractères spéciaux permettant d'indiquer des retours à la ligne ou des tabulations. En Python, on utilise le type de donnée *str* pour stocker les chaînes de caractères.

Le texte d'une chaine de caractères est placé entre guillemets simples ou doubles. Le texte d'une chaine de caractères peut être interprété comme une liste dont les éléments sont les caractères individuels qui le constituent. La plupart des opérations courantes sur les listes peuvent être utilisées sur les chaines de caractères à une différence importante près : On peut accéder aux caractères individuels d'une chaine de caractères par leur indice (parenthèses carrées), mais il n'est pas possible de modifier les caractères constituant les chaines par une affectation car les chaines de caractères sont un type immutable. Pour changer une chaine de caractères, il faut en fait en créer une nouvelle.

En Python 2, les chaines de caractères sont représentées à l'interne par des caractères ASCII sur 8 bits. Si l'on préfixe la chaine de caractère par un u, comme dans u'chaîne de caractères', la représentation interne des caractères se fera alors dans l'encodage Unicode 16 bits, permettant d'inclure sans problème les caractères spéciaux et accentués. Les caractères individuels ne sont cependant pas représentés par un type particulier comme en C mais par une chaine de caractères de longueur 1. En Python 3, la représentation en Unicode sur 16 bits est celle adoptée par défaut.

Le programme suivant définit un texte formatté en HTML et l'écrit sur la sortie standard.

```
html = "<html><body>TigerJython Web Site</body></html>"
print html
```

On peut sans problème parcourir une chaine caractère à caractère en utilisant une boucle *for* et un **indice** :

```
html = "<html><body>TigerJython Web Site</body></html>"

for i in range(len(html)):
   print html[i]
```

Il est cependant plus élégant d'utiliser une boucle *for* avec le mot-clé **in** pour en parcourir directement les éléments:

```
html = "<html><body>TigerJython Web Site</body></html>"
for c in html:
   print c
```

Une chaine de caractères peut également contenir des caractères de contrôle spéciaux appelés **séquences d'échappement** qui débutent par un **antislash** (*backslash* en anglais). Parmi celles-ci, on compte notamment le retour à la ligne *\n* et la tabulation *\t*. Ces séquences permettent donc de créer la chaîne de caractères présentée tout en début de chapitre:

```
html = "<html>\n   <body>\n      TigerJython Web Site\n   </body>\n</html>"
print html
```

Il est possible de lire un texte depuis un fichier texte. Pour cela, créer un fichier *welcome.html* à l'aide de votre éditeur de texte favori dans le même dossier que votre programme Python, en y mettant le contenu suivant:

```
<html>
   <body>
      <h1>TigerJython Web-Site</h1>
      Good morning
   </body>
</html>
```

On spécifie un titre de premier niveau avec la balise <h1>. Le programme suivant lit le fichier texte et stocke le contenu de ce dernier dans la chaine de caractères *html* qui est ensuite affichée sur la sortie standard

```
html = open("welcome.html").read()
print html
```

MEMENTO

Une chaine de caractères est un objet immutable constitué de caractères. On peut lire les caractères individuellement à l'aide de leur indice. Toutefois, toute tentative de modifier l'un des caractères de la chaine se soldera par une erreur (exception). En Python, contrairement à d'autres langages, il n'existe pas de type de donnée spécifique pour stocker les caractères individuels puisque des caractères isolés y sont considérés comme des chaines de caractères de longueur 1.

On peut ouvrir des fichiers à l'aide de la fonction *open()* en spécifiant le chemin du fichier à ouvrir. Ce chemin peut être relatif au dossier contenant l'archive *tigerjython2.jar* ou absolu si on le préfixe d'un slash (dans Windows, il faut également spécifier une lettre pour identifier le lecteur). Par exemple:

open("test/welcome.html")	*welcome.html* se trouve dans le sous-dossier *test* du dossier de *tigerjython2.jar*
open("/myweb/test/welcome.html")	*welcome.html* se trouve dans le dossier */myweb/test* du même lecteur contenant l'archive *tigerjython2.jar*
open("d:/myweb/test/welcome.html")	(Windows uniquement) *welcome.html* se trouve dans le dossier *\myweb\test* du lecture D:

Il est possible de joindre deux chaines de caractères avec l'opérateur + (concaténation). Il faut cependant que les deux opérandes soient réellement des chaînes de caractères. L'instruction

```
>>> "pi = " + 3.1459
```

engendrera donc une erreur. Il faut en effet commencer par convertir le nombre 3.1459 en chaine de caractères en utilisant la fonction *str()*:

```
>>> "pi = " + str(3.14159)
```

Les opérations les plus importantes sur les chaines de caractères:

s = "Python"	Définit une chaine (variante: *s = 'Python'*)
s[i]	Accède au caractère situé à la position d'indice *i*
s[start:end]	Nouvelle sous-chaine à partir des caractères situés entre *start* et *end* non compris
s[start:]	Nouvelle sous-chaine à partir des caractères depuis l'indice *start* jusqu'à la fin de la chaine
s[:end]	Nouvelle sous-chaine à partir des caractères depuis le début jusqu'à *end* non compris
s.index(x)	Indice de la première occurrence de x (-1 si absent)
s.find(x)	idem
s.find(x, start)	Indice de la première occurrence de x à partir de *start* (-1 si absent)
s.find(x,start,end)	Indice de la première occurrence de x entre *start* et *end* non compris (-1 si absent)
s.count(x)	Retourne le nombre d'occurrences de *x* dans *s*
x in s	Retourne *True* si la sous-chaine *x* est présente dans *s*
x not in s	Retourne *True* si la sous-chaine *x* n'est pas présente dans *s*
s1 + s2	Concaténation de *s1* et *s2* en une nouvelle chaîne
s1 += s2	Remplace *s1* par la concatenation de *s1* et *s2*
s * 4	Nouvelle chaine formée de 4 répétitions de la chaine *s*
len(s)	Retourne le nombre de caractères dans *s*

NAVIGATEUR WEB

Le rôle le plus fondamental d'un **navigateur Web** est d'interpréter les balises HTML présentes dans un document HTML et d'en effectuer le rendu dans une fenêtre en respectant au mieux les informations de structure présentes dans le balisage HTML. On peut effectuer le rendu du fichier *welcome.html* sur le PC après avoir installé un navigateur (Firefox, Explorer, Chrome, Safari, Opera, etc.).

TigerJython met à disposition une fenêtre de navigateur élémentaire comme instance de la classe *HtmlPane*. La méthode **insertText()** permet d'insérer un chaine contenant du code HTML et d'en effectuer un rendu basique.


```
from ch.aplu.util import HtmlPane

html = open("welcome.html").read()
pane = HtmlPane()
pane.insertText(html)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Un navigateur Web interprète le balisage d'un document HTML et en effectue le rendu d'après les informations de structure contenues dans le HTML.

HtmlPane ne connait que les balises HTML les plus courantes et fondamentales. Il n'est donc pas possible d'y afficher des documents HTML complexes. On peut également utiliser un objet *HtmlPane* pour afficher la sortie du programme dans une fenêtre séparée et avec une mise en forme moins austère que la simple console.

HYPERLIENS

La propagation explosive du web est essentiellement due au fait qu'un site Web peut contenir des éléments qui conduisent, par un simple clic de souris, à d'autres sites hébergé sur un autre serveur Web souvent situé à l'autre bout de la planète. Ces éléments sont appelés **hyperliens** (*hyperlink* en anglais) ou **liens hypertextes** et peuvent constituer un structure d'informations interconnectées, à la manière d'une toile d'araignée.

Créer à nouveau un fichier *welcome.html* en y incluant la balise <a>. On y utilise également la balise <p> qui définit un nouveau paragraphe précédé par défaut d'un retour à la ligne.

```
<html>
   <body>
      <h1>TigerJython Web-Site</h1>
      <p>Good morning!</p>
      <a href="http://www.tigerjython.ch/">TigerJython Home</a>
   </body></html>
```

Pour rendre les hyperliens utilisables, il est nécessaire de définir une fonction de rappel **linkCallback()**(ou tout autre nom similaire) et d'enregistrer celle-ci avec le paramètre *linkListener* du constructeur *HtmlPane()*. Cette fonction de rappel sera donc automatiquement appelée lors d'un clic sur le lien, ce qui va faire naviguer le *HtmlPane* vers l'URL contenue dans

l'attribut *href* de l'hyperlien.

```
from ch.aplu.util import HtmlPane

def linkCallback(url):
    pane.insertUrl(url)

html = open("welcomex.html").read()
pane = HtmlPane(linkListener = linkCallback)
pane.insertText(html)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Les hyperliens sont des références croisées dans un document permettant de sauter vers d'autres documents. Les documents liés constituent une fonctionnalité caractéristique du World Wide Web.Malheureusement, le rendu des pages Web n'est que partiel dans la fenêtre *HtmlPane*. Il est cependant possible d'utiliser le navigateur par défaut de la machine avec *HtmlPane.browse()* [**plus...**].

```
from ch.aplu.util import HtmlPane
HtmlPane.browse("www.tigerjython.com")
```

EXERCICES

1. La balise

   ```
   <img src="gifs/tigerlogo.png" width="120" height="116"></img>
   ```

   permet d'ajouter une image située dans le sous-dossier *gifs* du dossier contenant le fichier HTML. Les valeurs des attributs *width* et *height* devraient correspondre à la taille de l'image en pixels.

   Créer une fichier *showlogo.html* et un programme qui montre la page suivante dans une fenêtre *HtmlPane:*

   Vous pouvez télécharger l'image *tigerlogo.png* **ici**.

2. Définir les chaines *last_name, first_name, street* et *location* ainsi que *house_number* et *zip_code* contenant vos coordonnées personnelles ou des données imaginaires. Joindre ces différentes chaines de caractères en une seule chaine *address* en utilisant l'opérateur +, de telle manière que *print(address)* écrive sur la sortie standard l'information formatée de la manière suivante:
   first name, last name
   house number, street
   zip code, location

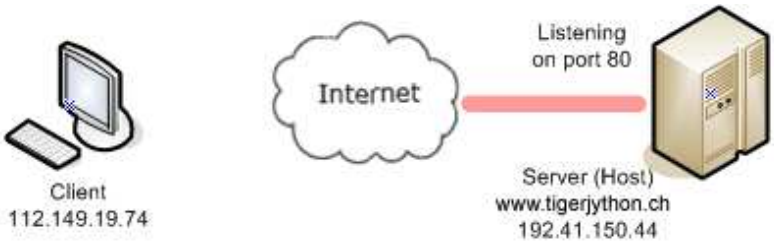
6.2 MODÈLE CLIENT-SERVEUR, PROTOCOLE HTTP

INTRODUCTION

Vous savez déjà qu'une page Web affichée dans le navigateur Web est décrite par un fichier texte ordinaire au format HTML qui est typiquement situé sur une machine distante, également appelée **hôte**. Pour être en mesure de localiser le fichier, le navigateur utilise une URL de la forme *http://hostname/filepath* structurée de la manière suivante:

HTTP	**H**yper**t**ext **T**ransfer **P**rotocol qui spécifie les conventions utilisées pour effectuer la communication entre le serveur Web et le client (navigateur)
hostname	Nom d'hôte de la machine sur laquelle tourne le serveur HTTP. Ce nom d'hôte permet d'identifier le serveur de manière unique sur Internet. Il s'agit soit d'une adresse IP sous la forme de 4 nombres compris entre 0 et 255, par exemple 192.41.150.141, ou d'un alias tel que **www.tigerjython.com**.
filepath	Le chemin d'accès au document HTML commence par un slash mais il est relatif au dossier racine du Site sur le serveur Web.

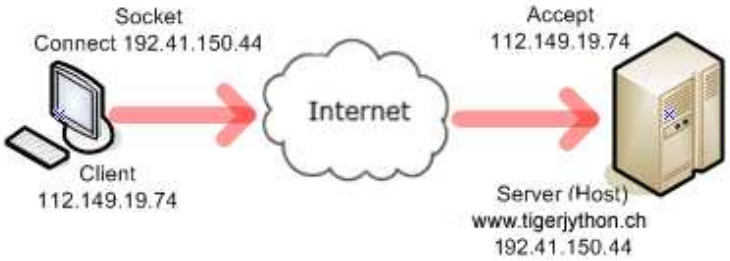
La communication entre le client et le serveur se déroule sur le mode requête-réponse (request-response) qui constitue l'un des principes les plus importants dans le domaine de la communication par ordinateur. Cela suppose que le serveur exécute un programme qui attend les requêtes des clients sur un port TCP spécifique (pour le Web, il s'agit par défaut du port 80).



L'échange de données comporte plusieurs phases distinctes détaillées dans les schémas ci-dessous:

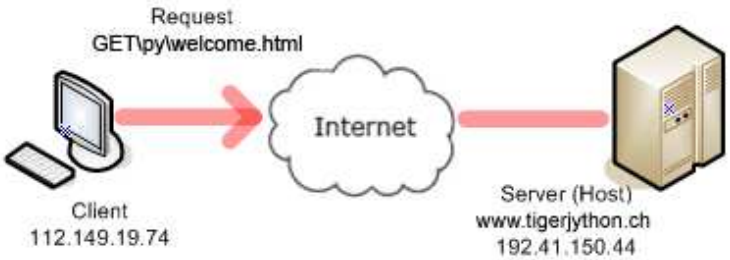
Phase 1:

Le client crée un socket par lequel il se connecte au serveur. Le serveur accepte la connexion et mémorise l'adresse IP du client.



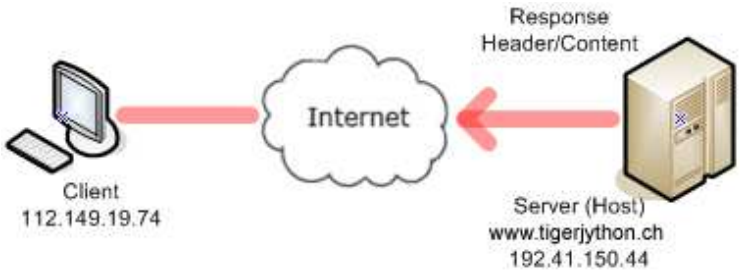
Phase 2:

Le client envoie une requête au serveur contenant le chemin d'accès à la ressource demandée.



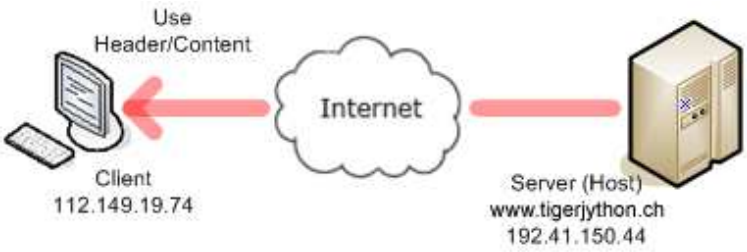
Phase 3:

Le serveur analyse et traite la requête puis envoie au client la ressource demandée, par exemple un fichier HTML..



Phase 4:

Le client (navigateur Web) reçoit la réponse et effectue le rendu de la page Web en l'affichant dans la fenêtre de navigation.



CONCEPTS DE PROGRAMMATION: *Hôte, client, adresse IP, Modèle requête – réponse, Protocole HTTP, Analyse syntaxique (parsing)*

REQUÊTE HTTP POUR OBTENIR UNE PAGE WEB

Le programme suivant effectue les phases 1, 2 et 4 pour obtenir le fichier *welcome.html* situé dans le sous-dossier *py* à la racine du serveur Web.

La méthode **socket()** de la classe *socket* stocke dans la variable *s* un objet socket. Cette méthode prend deux paramètres qui sont des constantes et qui indiquent le type de socket à créer.

```
import socket
import sys

host = "www.tigerjython.ch"
port = 80
remote_ip = socket.gethostbyname(host)

# Phase 1
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((remote_ip , port))
print "Socket Connected to " + host + " on ip " + remote_ip

# Phase 2
request = "GET /py/welcome.html HTTP/1.1\r\nHost: " + host + "\r\n\r\n"
s.sendall(request)

# Phase 4
reply = s.recv(4096)
print "\nReply:\n"
print reply
```

MEMENTO

Lorsqu'un navigateur Web (client HTTP) demande une page Web, il le fait en utilisant le protocole HTTP. Il s'agit d'une série de conventions, suivies par le client et le serveur, déterminant de manière très précise les procédures à utiliser pour l'échange de données. La commande (on parle de méthode dans le jargon du HTTP) GET est définie de la manière suivante dans la spécification du protocole:

Ligne 1	GET /py/welcomeex.html HTTP/1.1\r\n	Commande (méthode) permettant de rapatrier le fichier *welcome.html* situé dans le dossier dossier *py* à la racine du site, suivie de la version du protocole et des caractères spéciaux <retour chariot><retour à la ligne> (représentés par les séquences d'échappement \r\n)
Ligne 2	Host: hostname\r\n	Spécifie le nom d'hôte du serveur à contacter suivi des caractères spéciaux <retour chariot><retour à la ligne> représentés par les séquences d'échappement *\r\n*
Ligne 3	\r\n	Ligne blanche ne comportant que le saut de ligne *\r\n*

EN-TÊTES HTTP ET CONNECT

La réponse du serveur est constituée d'en-têtes servant à préciser des informations d'état ainsi que le contenu à proprement parler qui est le document HTML. Avant de pouvoir effectuer le rendu du code HTML reçu, il faut donc supprimer les informations d'en-têtes pour ne conserver que le code HTML et le transmettre à un *HtmlPane*.

```
import socket
import sys
from ch.aplu.util import HtmlPane

host = "www.tigerjython.ch"
port = 80
remote_ip = socket.gethostbyname(host)

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((remote_ip , port))
request = "GET /py/welcome.html HTTP/1.1\r\nHost: " + host + "\r\n\r\n"
s.sendall(request)
reply = s.recv(4096)

index = reply.find("<html")
html = reply[index:]

pane = HtmlPane()
pane.insertText(html)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

La fonction recv(4096) rend un maximum dem4096 caractŠres … partir d'une m,moire de donn,es dans lequel les caractŠres re‡us sont copi,s.

Pour supprimer les en-têtes de la réponse, on peut utiliser la méthode **find(str)** de la classe *str* permettant de rechercher la sous-chaine *sub_str* dans la chaîne *str* en retournant l'indice de la première occurrence. Si la sous-chaîne *sub_str* n'est pas trouvée dans *str*, la fonction retourne -1. Une fois que l'on connaît l'indice de la première occurrence de la sous-chaîne, on peut facilement supprimer tout ce qui la précède grâce à l'opérateur de slicing, *str[start :]* qui ne conserve que la partie de la chaine à partir de l'indice *start* jusqu'à la fin.

LECTURE DES PRÉVISIONS MÉTÉOROLOGIQUES

Vous vous demandez peut-être s'il est vraiment utile d'implémenter une procédure aussi compliquée pour afficher une page Web alors qu'il serait possible de n'écrire qu'une seule ligne ayant recours à la méthode *insertUrl()* du *HtmlPane*. Ce que vous avez appris par-là sera très utile si, au lieu de vouloir simplement afficher ce code HTML dans un *HtmlPane*, vous voulez en extraire une information bien précise. Le programme ci-dessous sert par exemple à lire le code HTML du site du bureau australien de météorologie pour en extraire les prévisions météorologiques actuelles.

Vous pouvez vous simplifier la vie encore davantage en utilisant la bibliothèque *urllib2* au lieu de créer explicitement le socket permettant de télécharger le code HTML du site [**plus...**].

Pour situer l'information désirée dans le code HTML, le programme d'analyse suivant représente le code HTML reçu à la fois dans la console de Python et dans le navigateur par défaut.

```
import urllib2
from ch.aplu.util import HtmlPane

url = "http://www.bom.gov.au/nsw/forecasts/sydney.shtml"
HtmlPane.browse(url)
html = urllib2.urlopen(url).read()
print html
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

L'utilisation d'une bibliothèque logicielle telle qu'*urllib2* simplifie grandement le code mais présente le désavantage d'occulter les mécanismes de base nécessaires à l'implémentation de la fonctionnalité.

ANALYSE SYNTAXIQUE DE TEXTES

Vous êtes maintenant confrontés à la tâche fort intéressante et non triviale consistant à extraire l'information pertinente d'une longue de chaîne de caractères : c'est ce qu'on appelle **l'analyse syntaxique** du texte.

Dans un premier temps, il faut supprimer toutes les balises HTML à l'aide d'une fonction *remove_html_tags()*. Il s'agit d'une procédure bien typique dont l'algorithme peut être décrit de la manière suivante:

On parcourt le texte caractère à caractère en mémorisant deux états distincts : ou bien le caractère lu se trouve à l'intérieur d'une balise HTML ou il se trouve à l'extérieur. Il faut uniquement copier les caractères de la chaine analysée si l'on ne se trouve pas à l'intérieur d'une balise ouvrante ou fermante. Les changements d'état se produisent lorsque l'on est en train de lire les caractères < ou > marquant le début ou la fin d'une balise.

Avant d'être en mesure de développer une fonctionnalité permettant d'extraire l'information voulue, il faut analyser le contenu du texte après épuration des balises HTML. Cela consiste essentiellement à copier le texte dépourvu des balises HTML dans un éditeur de texte et à chercher une séquence de caractères marquant de manière unique le début de l'information recherchée ainsi qu'une séquence de caractères marquant la fin de cette information. La méthode *str.find()* sera alors d'un grand secours pour trouver l'indice *start* marquant le début de l'information. Pour trouver l'indice *end* marquant la fin de l'information, on réutilise *str.find()* en recherchant à partir de *start*. Pour ce site Web, la séquence de caractères du début est « Sydney area » et la séquence de fin est « Summary ». Le texte se trouvant entre les deux est extrait par une opération de slicing *[start:end]*.

```
import urllib2

def remove_html_tags(s):
    inTag = False
    out = ""

    for c in s:
        if c == '<':
            inTag = True
        elif c == '>':
          inTag = False
        elif not inTag:
            out = out + c
    return out

url = "http://www.bom.gov.au/nsw/forecasts/sydney.shtml"
html = urllib2.urlopen(url).read()
html = remove_html_tags(html)

start = html.find("Sydney area")
end = html.find("Summary", start)
html = html[start:end].strip()

print html

from soundsystem import *

initTTS()
selectVoice("english-man")
sound = generateVoice(html)
openSoundPlayer(sound)
play()
```

MEMENTO

The parsing of texts is usually done character by character. In many cases, however, methods of the string class may help as well. En HTML, les caractères spéciaux et accentués sont encodés de manière spéciale à l'aide de séquences appelées entités HTML débutant par une esperluette & et se terminant par un point-virgule ; selon la table suivante:

<	<	ê	ê
>	>	ë	ë
à	à	î	î
á	á	ï	ï
â	â	ô	ô
æ	æ	œ	œ
ç	ç	ù	ù
è	é	û	û
é	é	ÿ	ÿ

SYNTHÈSE VOCALE / LECTURE DE LA MÉTÉO

En utilisant les connaissances acquises dans un précèdent chapitre portant sur le son, vous êtes en mesure de créer un programme qui va utiliser la synthèse vocale pour lire automatiquement les prévisions météorologiques en quelques lignes de code:

```
from soundsystem import *

initTTS()
selectVoice("german-man")
```

```
sound = generateVoice(html)
openSoundPlayer(sound)
play()
```

EXERCICES

1. On trouve à l'adresse *http://www.timeanddate.com* de nombreuses informations intéressantes pouvant être extraites et réutilisées par un programme personnel. On peut par exemple en extraire le bulletin météo de n'importe quelle ville dans le monde. Visitez l'adresse *http://www.timeanddate.com/weather/canada/halifax* avec navigateur Web. Elle permet d'obtenir la température d'une ville choisie de manière arbitraire.

   Déroulement : Sauver en intégralité le texte reçu lors de la visite de l'URL indiquée précédemment et y rechercher la température. Écrire ensuite un programme qui extrait cette valeur en utilisant de manière appropriée les méthodes de la classe *str*. Le programme doit permettre à l'utilisateur de sélectionner un pays et une ville à l'aide de boîtes de dialogue *inputString()* ou *EntryDialog()* et d'écrire la température du lieu en question dans la console ou dans un champ *StringEntry* de la boîte de dialogue.

2*. La méthode *urllib2.urlopen(url)* lève une exception si l'URL n'existe pas. Si l'on enrobe cet appel d'une structure *try-except* permet de gérer l'erreur de manière douce en exécutant le bloc de la branch *except* en cas d'erreur au lieu de faire planter le programme.

```
try:
   urllib2.urlopen(url)
except:
   print "Error"
```

   Modifier le programme de l'exercice 1 pour qu'il affiche un message d'erreur pertinent si la ville n'existe pas au lieu de bêtement se planter

3. On peut visualiser la requête GET effectuée par un navigateur Web récent à l'aide des outils développeurs. Dans Firefox, on peut les ouvrir à l'aide du raccourci (Ctrl+Maj+I) puis se rendre dans l'onglet « Réseau » et rafraîchir la page. On peut retrouver les lignes de la requête GET envoyée dans la section « En-têtes de la requête » (HTTP headers) dont voici un extrait lorsqu'on visite la page Wikipedia portant sur le protocole HTTP:

```
GET /wiki/Hypertext_Transfer_Protocol HTTP/1.1
Host: en.wikipedia.org
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:41.0) Gecko/20100101
            Firefox/41.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: fr,fr-FR;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Cookie: WMF-Last-Access=24-Oct-2015; GeoIP=CH:::47.00:8.00:v4;
        enwikimwuser-sessionId=9c0945c3c63ca081
Connection: keep-alive
If-Modified-Since: Thu, 22 Oct 2015 09:10:48 GMT
Cache-Control: max-age=0
```

   Les lignes 3 à 12 sont des en-têtes HTTP facultatifs faisant partie de la requête pour donner de plus amples informations au serveur sur le contenu attendu, le type de navigateur effectuant la requête, …
   Consigne : Essayer de comprendre chaque en-tête HTTP envoyé par le navigateur en utilisant si nécessaire la documentation du protocole HTTP. Relever les en-têtes qui peuvent être problématiques du point de vue de la protection de la vie privée.

6.3 RECHERCHE BING, DICTIONNAIRES

INTRODUCTION

Il est possible d'utiliser des moteurs de recherche connus tels que Google, Bing ou Yahoo pour effectuer des **recherches programmatiques** sur le Web. Au lieu de taper des mots-clés dans un formulaire sur le site Web du moteur de recherche, il s'agit d'effectuer par programme une requête HTTP GET sur une URL du site en fournissant des paramètres supplémentaires. Ces données sont ensuite évaluées par une **application Web**, c'est-à-dire un programme exécuté par le serveur Web chargé de les traiter et de retourner les résultats sous forme de réponse HTTP [**plus...**].

De plus, certains moteurs de recherche mettent à disposition un service de recherche utilisable par l'intermédiaires d'une API (Application Programming Interface). Bien que ces services soient généralement payants, ils sont parfois disponibles gratuitement en version restreinte pour faciliter le développement d'applications ainsi que la prise en main de l'API. L'API du moteur de recherche Bing est par exemple disponible en version limitée et permet ainsi de créer soi-même une interface graphique personnalisée pour lancer des recherches Bing.

Les API des moteurs de recherche retournent les données dans un format spécial appelé JSON (JavaScript Object Notation). Le module Python *json* permet de convertir facilement du JSON en un dictionnaire Python. De ce fait, pour être en mesure d'extraire les informations retournées en JSON par le moteur de recherche Bing, il faudra d'abord savoir précisément ce que sont les dictionnaires Python et comment on les manipule.

CONCEPTS DE PROGRAMMATION: *Application Web, Dictionnaires Python*

COMPRENDRE LES DICTIONNAIRES

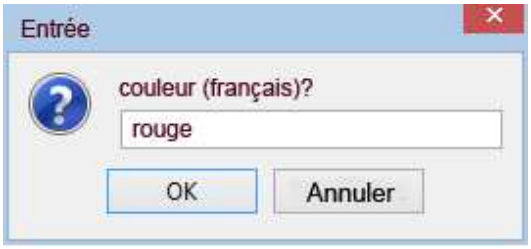
Comme son nom l'indique, un dictionnaire est une structure de données similaire à un dictionnaire de langue. On peut imaginer des paires de mots formées d'un mot connu à gauche et d'un mot de la langue étrangère à droite en excluant toute ambiguïté : on imagine que tout mot de la langue d'origine comporte une unique signification dans la langue étrangère (ce qui est faux en réalité). L'exemple ci-dessous montre la correspondance entre les noms de couleurs en français et en anglais :

Français	**Englisch**
bleu	blue
rouge	red
vert	green
jaune	yellow

(Dans un vrai dictionnaire, les mots sont cependant arrangés dans l'ordre alphabétique afin de simplifier la recherche d'un mot spécifique.)

Le mot de la colonne de gauche est la **clé** (*key*) et le mot de droite est la valeur (*value*). Un dictionnaire contient donc des paires **clé-valeur**. Les valeurs peuvent prendre n'importe quel type de données et les clés n'importe quel type de données immutable. Ceci exclut la possibilité d'utiliser des listes ou dictionnaires en tant que clé d'un dictionnaire mais autorise en revanche les nombres, les chaines de caractères et les tuples.

Le programme ci-dessous traduit du français vers l'anglais les couleurs mentionnées dans le tableau précédent. Si le mot saisi n'est pas une clé du dictionnaire (colonne de gauche), l'erreur est gérée et un message est affiché dans la console.




```
dict = {"bleu":"blue", "rouge":"red", "vert":"green", "jaune":"yellow"}

print "All entries:"
for key in dict:
    print key + " -> " + dict[key]

while True:
    color = input("couleur (français)?")
    if color in dict:
        print color + " -> " + dict[color]
    else:
        print color + " -> " + "(not translatable)"
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Un dictionnaire est un ensemble de paires clé-valeur. À la différence des listes, ces paires ne sont pas ordonnées. Pour définir un dictionnaire, on utilise des accolades {}, on sépare les paires par des virgules et les clés des valeurs par un double-point de la manière suivante :
>>> dictionnaire = {clé1 : valeur1, clé2 : valeur2, …}

Important operations:

dict[key]	Retourne la valeur associée à la clé *key* si la clé existe
dict[key] = value	Ajoute une nouvelle paire clé-valeur ou change la valeur associée à la clé *key*
len(dict)	Retourne le nombre de paires clé-valeur présentes dans le dictionnaire
del dict(key)	Supprime la paire dont la clé est *key*
key in dict	Retourne *True* si la clé *key* existe
dist.clear()	Supprime toutes les paires. Il reste donc un dictionnaire vide

On peut parcourir un dictionnaire à l'aide d'une boucle *for* de la manière suivante

```
for key in dict:
```

LE DICTIONNAIRE, UNE STUCTURE DE DONNÉES PERFORMANTE

Vous vous demandez peut-être pourquoi il faut une nouvelle structure de données pour stocker des paires clé-valeur alors qu'on pourrait très bien les stocker dans une liste dont les éléments seraient de courtes listes de deux éléments [clé, valeur].
Le gros avantage des dictionnaires réside dans la facilité d'accès aux paires sur la base de la clé à l'aide des crochets carrés. En d'autres termes, il est très facile de vérifier si une information s'y trouve ou non et de la retourner le cas échéant. Il ne s'agit pas simplement d'une facilité de

langage mais d'un réel gain de performance par rapport à une liste. L'avantage des dictionnaires sur les listes pour trouver une information devient très évident lorsqu'il y a des centaines, voire des milliers ou des millions de paires clé-valeur.

Comme application intéressante, le programme ci-dessous retourne le numéro postal de n'importe quelle ville suisse. Les données se trouvent dans le fichier texte ***chplz.txt*** disponible en téléchargement grâce au lien précédent. Copiez ce fichier dans le même dossier que votre programme Python. Le fichier contient sur chaque ligne du fichier une association ville-NPA et ne comporte aucune ligne vide, même tout à la fin :

Aarau:5000
Aarburg:4663
Aarwangen:4912
Aathal Seegraeben:8607
...

La première étape consiste à charger ce fichier dans la mémoire en créant un dictionnaire contenant les paires ville(clé)-NPA(valeur). On utilise à ce dessein la fonction **read()** pour le charger en tant que chaine de caractères dans la variable *plzStr* qui sera ensuite séparée ligne par ligne grâce à la méthode split("\n") [**plus...**].

Pour construire le dictionnaire, il faut séparer la clé (ville) et la valeur (npa) de chaque ligne sur la base du double-point et les ajouter à l'aide des crochets carrés au dictionnaire *plz* qui est initialement vide. Une fois le dictionnaire construit, il est possible d'accéder facilement à un numéro postal en indiquant la ville désirée entre crochets carrés.

```
file = open("chplz.txt")
plzStr = file.read()
file.close()

pairs = plzStr.split("\n")
print str(len(pairs)) + " pairs loaded"
plz = {}

for pair in pairs:
    element = pair.split(":")
    plz[element[0]] = element[1]

while True:
    town = input("City?")
    if town in plz:
        print "The postal code of " + town + " is " + str(plz[town])
    else:
        print "The city " + town + "was not found."
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Il est vraiment très facile et très rapide d'accéder à la valeur correspondant à une certaine clé dans un dictionnaire [**plus...**].

UTILISER BING POUR SON PROPRE PROFIT

Le programme ci-dessous utilise le moteur de recherche Bing pour rechercher des sites correspondant à la requête saisie par l'utilisateur et afficher les résultats retournés. Pour pouvoir accéder au service de recherche Bing, il faut disposer d'une clé d'authentification personnelle qu'il faut se procurer de la manière suivante:

Visiter la page **https://www.microsoft.com/cognitive-services/en-us/apis** et choisir *"Get started for free"*. Vous allez devoir fournir le mot de passe de votre compte Microsoft ou en créer un nouveau. Dans la page intitulée *Microsoft Cognitive Services*, sélectionner *"APIs"* et *"Bing Web Search"* puis cliquer sur *"Request new trials"*. Descendre dans la page et choisir *"Search Bing-Free"*. Après avoir confirmé en cliquant sur *"Subscribe"*, vous recevrez deux valeurs. Sauvegardez l'une d'elles par copier-coller dans un fichier texte car vous en aurez bientôt besoin. Il est également possible de retrouver ces clés sous votre compte Microsoft.

Le programme ci-dessous procède en envoyant une requête HTTP GET dans laquelle figure la requête de la recherche. La réponse de Bing est une chaine de caractères dans laquelle l'information est structurée par des accolades, selon le format JSON. La méthode **json.load()** permet de convertir cette chaine de caractères en un dictionnaire Python qui peut être analysé plus efficacement. Durant la phase de test, il peut être utile de bien étudier les différentes imbrications des données en imprimant la réponse JSON vers la console. Ces lignes peuvent être supprimées ou commentées par la suite. Quelles pages Bing trouve-t-il pour la requête "Hillary Clinton" ?

```
import urllib2
import json

def bing_search(query):
    key = 'xxxxxxxxxxxxxxxxxxxxx' # use your personal key
    url = 'https://api.cognitive.microsoft.com/bing/v5.0/search?q=' + query
    urlrequest = urllib2.Request(url)
    urlrequest.add_header('Ocp-Apim-Subscription-Key', key)
    responseStr = urllib2.urlopen(urlrequest)
    response = json.load(responseStr)
    return response

query = input("Enter a search string(AND-connect with +):")
results = bing_search(query)
#print "results:\n" + str(results)
webPages = results['webPages']
print "Number of hits:", webPages["totalEstimatedMatches"]
print "Found URLs:"
values = webPages.get('value')
for item in values:
    print item["displayUrl"]
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Comme le montre cet exemple, un dictionnaire peut contenir d'autres dictionnaires comme valeurs, ce qui permet de créer des structures d'information hiérarchiques de manière similaire au XML.

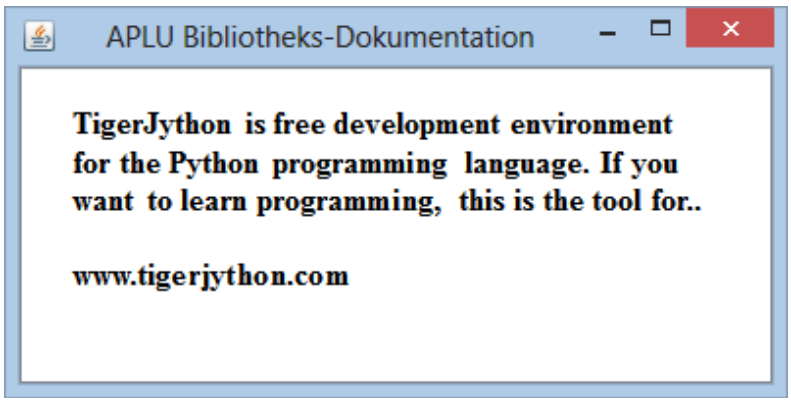
La clé d'authentification est utilisée dans un en-tête HTTP supplémentaire de la requête GET. Il est possible de personnaliser la recherche Bing en spécifiant des paramètres additionnels. Par exemple, en ajoutant la chaine "&count=20" à l'URL, le nombre de résultats est limité à 20. Pour plus d'informations, consulter la **référence de l'API**.

EXERCICES

1. Améliorer de manière incrémentale le programme permettant de retrouver le code postal d'une ville suisse même dans les cas suivants :

   a. Dans le champ de saisie, le nom de la ville peut être précédé et suivi de caractères d'espacement.
   b. Le nom de la ville est saisi en mélangeant majuscules et minuscules
   c. Les trémas des noms de villes germanophones sont remplacés par les séquences Ae, Oe, ae, oe, et ue
   d. Les accents sont omis des noms des villes (ceci engendre une ambigüité pour ö)
   e. Certaines localités portent des noms ambigus mais disposent d'informations supplémentaires. Proposer un moyen de s'en sortir avec ces cas spéciaux.

2. Effectuer une recherche Bing pour écrire le titre et le contenu du résultat disposant du score de pertinence le plus élevé dans un *HtmlPane*. Par exemple, pour la requête de recherche « tigerjython », le résultat devrait ressembler à la capture ci-dessous :



chapitre sept

JEUX & POO

Objectifs d'apprentissage

- Être capable de définir en Python et d'utiliser une classe comportant un constructeur, des variables d'instance et des méthodes.
- Comprendre ce qu'est une hiérarchie de classes et être en mesure de définir et d'utiliser des classes dérivées.
- Être capable d'expliquer en termes simples ce qu'est le polymorphisme.
- Avoir une compréhension basique de la conception de la bibliothèque de jeux *JGameGrid* et être en mesure de l'utiliser pour réaliser des jeux vidéo simples.

« De nos jours, on ne peut songer aux médias digitaux sans penser aux jeux vidéo. Du fait de la grande importance des jeux pour les adolescents, de nombreux éducateurs sont poussés à explorer le grand potentiel didactique qu'ils représentent. L'apprentissage par le jeu (GBL = Game-Based Learning) fait l'objet de nombreuses étude scientifiques à tel point que ce domaine est maintenant intégré aux programmes d'étude actuels. Dans les cours d'informatique, les jeux peuvent être abordés du point de vue des producteurs. Les jeux étant des programmes hautement dynamiques, ils encouragent l'apprenant à penser en termes de procédures. De plus, les objets graphiques (sprites) dont ils regorgent sont extrêmement bien adaptés pour l'introduction à la programmation orientée objets. »

Jarka Arnold, Aegidius Plüss
in "Games as an Introduction to Object-Oriented Programming"

7.1 DES OBJETS, ENCORE DES OBJETS

INTRODUCTION

Dans notre quotidien, nous sommes entourés d'une multitude d'objets. Du fait que les logiciels sont souvent conçus à l'image de la réalité, il est très naturel d'introduire la notion d'objets en informatique. C'est ce qu'on appelle la programmation orientée objets (POO). Depuis plusieurs décennies, la notion de POO s'est révélée être une révolution dans le génie logiciel à tel point que pratiquement n'importe quel logiciel significatif est actuellement développé en utilisant cette technique [**plue...**]. In the following chapter you will learn the main concepts of OOP so that you can participate in the hype.

Nous avons déjà vu que les tortues sont représentées par des objets. Une tortue possède certaines **propriétés** comme sa couleur, sa position et son angle de visée ainsi que certains **comportements**, comme la capacité d'avancer, de tourner, etc. En POO, des objets possédant des propriétés et des comportements communs sont regroupés par **classes**. Les objets tortues appartiennent tous à la classe *Turtle* : on dit en termes techniques qu'ils sont des **instances** de la classe *Turtle*. Afin de créer un objet, il faut utiliser une classe prédéfinie ou **définir une nouvelle classe**.

En termes techniques, les propriétés des objets sont également appelées **attributs** ou **variables d'instance** et les capacité sont souvent appelés **méthodes** ou **opérations**. Ces attributs et méthodes sont en fait des variables et des fonctions à l'exception qu'elles sont encapsulées dans la classe. Pour s'y référer de l'extérieur de la classe, il suffit de préfixer leur nom par celui d'une instance de classe et de **l'opérateur point**.

CONCEPTS DE PROGRAMMATION: *Classe, objet (instance), propriété, capacité, attribut / variable d'instance, méthode, héritage, classe de base (parente), constructeur*

UNE FENÊTRE DE JEU ADAPTIVE

Développer un jeu vidéo sans recourir à la programmation orientée objets est un véritable supplice de Sisyphe du fait que les acteurs du jeu et tous les objets qui interviennent dans leur environnement sont justement des objets en interaction. Dans un jeu en 2D, le plateau de jeu est une fenêtre rectangulaire représentée par la classe *GameGrid* de la bibliothèque *JGameGrid*. TigerJython fabrique une instance de cette classe *GameGrid* lors de l'appel de la fonction **makeGameGrid()** et affiche la fenêtre lors d'un appel à la fonction **show()**. Il est possible de personnaliser l'apparence de la fenêtre de jeu avec des paramètres appropriés.

L'appel *makeGameGrid(10, 10, 60, Color.red, "sprites/town.jpg", False)* aura par exemple pour effet d'afficher une fenêtre de jeu de taille 10x10 cellules carrées ayant chacune 60 pixels de côté. Le paramètre *Color.red* indique la couleur du quadrillage et le cinquième paramètre indique qu'il faut utiliser **town.jpg** en guise d'image de fond. Le dernier paramètre booléen désactive la barre de navigation, ce qui n'est pas nécessaire dans notre cas.

```
from gamegrid import *

makeGameGrid(10, 10, 60, Color.red, "sprites/town.jpg", False)
show()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Les méthodes de la classe *GameGrid* sont disponibles en tant que fonctions lorsque l'on crée la fenêtre de jeu avec *makeGameGrid()*. On peut cependant également créer soi-même une instance manuellement et appeler les méthodes à l'aide de l'opérateur point.

```
from gamegrid import *

gg = GameGrid(10, 10, 60, Color.red, "sprites/town.jpg", False)
gg.show()
```

La fenêtre de jeu est constituée de 10x10 cellules carrées dont la taille est de 60 pixels. Puisque les lignes de la grille sont également affichées tout en bas et tout à droite, la fenêtre a en fait une taille de 601 x 601 pixels. Cela correspondant à la taille minimale de l'image d'arrière-plan.

Le dernier paramètre booléen détermine s'il faut afficher une barre de navigation.

DÉFINIR UNE CLASSE PAR DÉRIVATION (HÉRITAGE)

Lorsque l'on définit une classe, on peut choisir si celle-ci est indépendante des autres ou si, au contraire, il s'agit d'une **classe dérivée** d'une classe déjà existante. Toutes les propriétés et méthodes de la classe parente aussi appelée classe de base ou superclasse sont disponibles dans la classe fille (dérivée). Autrement dit, la classe dérivée (ou **sous-classe**) **hérite** des propriétés et des méthodes de ses classes parentes.

Dans la classe *JGameGrid,* les personnages de jeu sont appelées **acteurs** et sont des instances de la classe prédéfinie *Actor*. Pour définir son propre personnage, si suffit de définir une classe dérivée de la classe *Actor*.

La définition d'une classe débute pas l'usage du mot-clé **class** suivi du nom attribué à cette nouvelle classe ainsi qu'une paire de parenthèses. Entre parenthèses, on note le nom d'une ou plusieurs classes qui feront office de classes parentes. Comme on veut dériver notre personnage de la classe *Actor*, c'est ce que l'on indiquera entre parenthèses suivants le nom de la classe.

La définition de la classe contient la définition de ses méthodes qui ne sont rien d'autre que des fonctions dont la seule particularité est de prendre **self** comme premier paramètre. Ce paramètre *self* permet d'accéder, depuis le code encapsulé dans la classe, aux autres méthodes et variables d'instance présentes au sein de l'objet.

On débute généralement la définition d'une classe par une **méthode spéciale** *__init__(self, …)* reconnaissable aux doubles caractères de soulignement qui marquent le début et la fin du *init*. Cette méthode spéciale, nommée **constructeur,** est invoquée automatiquement lors de la création d'un objet de la classe concernée. Dans le programme ci-dessous, on appelle le constructeur de la classe de base *Actor* depuis le constructeur de la classe *Alien* à laquelle il faut spécifier le chemin d'accès au sprite choisi.

On définit ensuite la méthode ***act()*** qui joue une rôle central dans l'animation du jeu puisqu'elle est invoquée par le gestionnaire du jeu lors de chaque cycle de simulation (déplacements). Il s'agit là d'une astuce particulièrement intelligente puisqu'elle permet de ne plus se soucier des animations au sein d'une structure de répétition comme une boucle *while*.

La méthode **act()** permet de programmer le comportement d'un acteur à chaque cycle du jeu.

Dans notre cas, on ne fait que déplacer l'acteur sur une autre case de la grille avec la fonction **move()**. Du fait que *move()* est une méthode héritée de la classe de base *Actor*, elle fait partie intégrante des instances de la classe *Alien*, ce qui nous amène à l'invoquer en la préfixant de *self*.

Une fois la classe *Alien* définie, on crée un objet de la classe *Alien* avec
>>> alien_object = Alien()

L'avantage de la POO est qu'il est possible de **créer** ainsi autant d'aliens (instances de la classe *Alien*) que nécessaires sans que les données de ceux ne viennent à entrer en conflit dans le programme. Chaque objet a en effet sa propre **individualité**, comme dans la réalité, de sorte que chaque alien sait exactement comme se déplacer lors de chaque pas de simulation grâce à sa méthode *move()*.

On utilise la fonction **addActor()** pour ajouter au plateau de jeu chacun des aliens générés en spécifiant les coordonnées de sa position de départ dans la grille. La cellule de coordonnées (0,0) est située en haut à gauche de la grille de jeu et l'axe *Oy* est orienté vers le bas. Pour lancer le cycle de simulation, il faut appeler la fonction **doRun()**.

```
from gamegrid import *

# ---------------- class Alien ----------------
class Alien(Actor):
    def __init__(self):
        Actor.__init__(self, "sprites/alien.png")

    def act(self):
        self.move()

makeGameGrid(10, 10, 60, Color.red, "sprites/town.jpg", False)
spin = Alien() # object creation, many instances can be created
urix = Alien()
addActor(spin, Location(2, 0), 90)
addActor(urix, Location(5, 0), 90)
show()
doRun()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

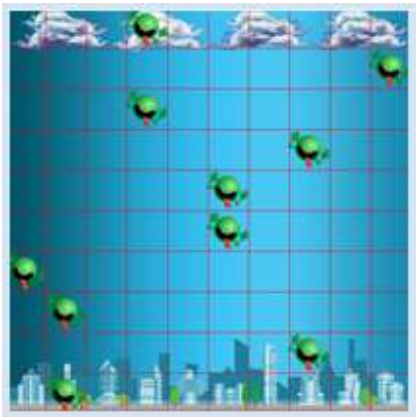
MEMENTO

La définition d'une classe débute par le mot-clé **class** et encapsule les méthodes ainsi que les variables d'instance de la classe. Le constructeur de la classe, nommé **__init__** est appelé automatiquement par Python lors de la création des objets (instanciations de la classe). Pour créer un objet (ou instance), il faut écrire le nom de la classe et, entre parenthèses, spécifier les arguments demandés par le constructeur.

Tous les personnages de jeu sont dérivés de la classe *Actor* et leur comportement lors de chaque cycle de simulation est personnalisé dans la méthode dans la méthode **act()**.

La fonction **addActor()** permet d'ajouter un personnage au plateau de jeu à la position et avec l'angle de départ indiqués en paramètres. L'angle 0 indique une orientation à l'Est (vers la droite) et le **sens positif correspond au sens des aiguilles de la montre**.

UNE ATTAQUE D'ALIENS



Vous avez pu observer les avantages considérables du paradigme orienté objets en voyant avec quelle facilité on peut peupler le plateau de jeu d'une myriade d'aliens tombant du ciel et cela en très peu de lignes de code.

Dans le programme ci-dessous, on fait en sorte que toutes les 0.2 secondes, un nouvel alien soit créé aléatoirement dans une des cellules de la première ligne du haut.

```
from gamegrid import *
import random

# ---------------- class Alien ----------------
class Alien(Actor):
    def __init__(self):
        Actor.__init__(self, "sprites/alien.png")

    def act(self):
        self.move()

makeGameGrid(10, 10, 60, Color.red, "sprites/town.jpg", False)
show()
doRun()

while not isDisposed():
    alien = Alien()
    addActor(alien, Location(random.randint(0, 9), 0), 90)
    delay(200)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Dans le programme principal, afin de garantir une fermeture propre, il faut une boucle qui teste à chaque itération la valeur booléenne retournée par **isDisposed()** pour savoir si la fenêtre de jeu a été fermée.

Note: Il est parfois nécessaire de fermer TigerJython et de le rouvrir afin de permettre aux sprites et aux images d'arrière-fond de se recharger correctement en cas de modification des fichiers images impliqués.

SPACE INVADERS LIGHT

Dans le premier jeu vidéo que vous allez développer, le joueur doit tenter de repousser une invasion d'aliens en les éliminant d'un clic de souris. Le joueur perd un point par alien qui atterrit avec succès dans la ville.

Pour intégrer le support de la souris dans le programme, il faut ajouter une fonction de rappel *pressCallback* et l'enregistrer avec le paramètre nommé *mousePressed*. Cette fonction de rappel commence par obtenir les coordonnées grille du clic en examinant l'objet *e* reçu en paramètre pour y trouver dans quelle cellule le clic a été effectué. Si cette cellule est occupée par un alien, celui-ci sera retourné par **getOneActorAt()** tandis que si elle est vide, la valeur *None* sera retournée. La fonction. **removeActor()** supprime l'acteur du plateau de jeu.

```
from gamegrid import *
import random

# ---------------- class Alien ----------------
class Alien(Actor):
    def __init__(self):
        Actor.__init__(self, "sprites/alien.png")

    def act(self):
        self.move()

def pressCallback(e):
    location = toLocationInGrid(e.getX(), e.getY())
    actor = getOneActorAt(location)
    if actor != None:
        removeActor(actor)
    refresh()

makeGameGrid(10, 10, 60, Color.red, "sprites/town.jpg", False,
             mousePressed = pressCallback)
setSimulationPeriod(800)
show()
doRun()

while not isDisposed():
    alien = Alien()
    addActor(alien, Location(random.randint(0, 9), 0), 90)
    delay(1000)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Du fait que *act()* est appelée une fois par cycle de simulation, la vitesse d'exécution du jeu sera très influencée par ce petit paramètre dont la valeur par défaut est 200 ms. On peut néanmoins changer cette valeur à l'aide de la fonction **setSimulationPeriod()**.

Le rendu du plateau de jeu est reconstruit entièrement à chaque pas de simulation, ce qui implique un court laps de temps de latence entre le moment ou l'état du jeu subit une modification et le moment où le rendu est effectué à l'écran. Si l'on souhaite effectuer le rendu immédiatement lors d'un clic de souris, on peut le lancer manuellement à l'aide de la fonction **refresh()**.

EXERCICES

1. Créer sa propre image de fond à l'aide d'un éditeur d'images puis l'ajouter au dossier *sprites*, lui-même présent soit dans le même dossier que le script Python, soit dans le dossier *<userhome>/gamegrid/sprites*). Il est également possible de spécifier un chemin d'accès absolu.

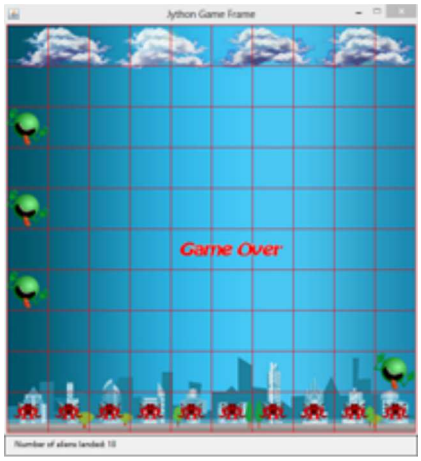
2. Ajouter une barre de statut de 30 pixels de haut avec l'appel *addStatusBar(30)* et y indiquer à l'aide de *setStatusText()* le nombre d'aliens qui ont atterri dans la ville malgré la vigilance du joueur.

3. Les aliens, une fois atterris, ne devraient pas simplement disparaître mais être remplacés par un sprite différent et immobile qui apparaît dans la cellule d'atterrissage. Le sprite "sprites/alien_1.png" fera par exemple très bien l'affaire.

   Conseil: avec *removeSelf()*, il est possible de supprimer l'acteur pour le remplacer par un nouvel acteur créé avec *addActor().*

4*. Les aliens qui sont atterris communiquent aux aliens attaquants leur position d'atterrissage de sorte que les futurs aliens ne tombent que dans les colonnes non encore occupées. Une fois que toutes les colonnes ont été conquises, le jeu se termine avec le message « Game Over ». On peut utiliser à cet effet le sprite ("sprites/gameover.gif").

   (Conseil: on peut mettre le gestionnaire de jeu en pause avec la fonction *doPause()*)



5*. Étendez le jeu en y ajoutant des nouveautés de votre propre cru.

7.2 CLASSES ET OBJETS

INTRODUCTION

.

Vous avez déjà pu vous familiariser avec les notions essentielles de la programmation orientée objets (POO) et compris qu'il serait très difficile de développer un jeu vidéo en Python sans recourir à cette technique. Il est de ce fait capital de continuer à comprendre ces notions ainsi que leur implémentation en *Python* de manière plus systématique.

CONCEPTS DE PROGRAMMATION: *Héritage, hiérarchie de classes, redéfinition (overriding), relation « is-a », héritage multiple*

VARIABLES D'INSTANCE

Les animaux se prêtent bien à être représentés par des objets. On commence par définir une classe *Animal* qui affiche l'image de l'animal approprié dans l'arrière-fond de la fenêtre de jeu. Lors de la construction d'une instance de cette classe, on spécifie au constructeur le chemin vers le fichier image à utiliser de sorte que le méthode *showMe()* soit capable d'afficher l'image grâce aux méthodes de dessin de la classe *GGBackground.*

Les animaux se prêtent bien à être représentés par des objets. On commence par définir une classe *Animal* qui affiche l'image de l'animal approprié dans l'arrière-fond de la fenêtre de jeu. Lors de la construction d'une instance de cette classe, on spécifie au constructeur le chemin vers le fichier image à utiliser de sorte que le méthode *showMe()* soit capable d'afficher l'image grâce aux méthodes de dessin de la classe *GGBackground.* Le constructeur qui reçoit ce chemin vers le fichier image doit le sauver dans une variable de sorte que toutes les autres méthodes de la classe puissent y avoir accès ultérieurement. Une telle variable est un attribut d'instance ou variable d'instance. En *Python*, les variables d'instance sont préfixées au sein des objets par *self* et sont créées en mémoire lorsqu'on leur affecte pour la première fois une valeur. Comme nous l'avons déjà mentionné, le constructeur porte le nom spécial **__init__**(avec deux caractères de soulignement « underscore » avant et après). Aussi bien le constructeur que les méthodes de la classe doivent impérativement être définis avec **self** comme premier paramètre formel, sans quoi des erreurs apparaîtront lors de leur utilisation.

On commence donc par définir le constructeur comme suit :

```
def __init__(self, imgPath:
```

de même qu'une méthode

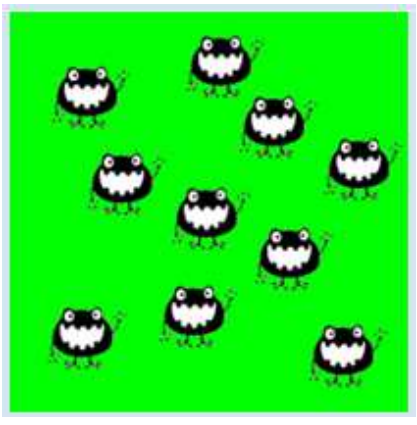
```
def showMe(self, x, y:
```

Une fois que l'on a créé un objet *myAnimal* avec

```
myAnimal = Animal(bildpfad)
```

Une fois que l'on a créé un objet *myAnimal* avec

```
myAnimal.showMe(x, y)
```



Il est particulièrement indiqué de recourir à la POO lorsque l'on utilise plusieurs objets de la même classe. Pour bien mettre ceci en évidence, le programme ci-dessous fait apparaitre un nouvel animal lors de chaque clic de souris.

```
from gamegrid import *

# ---------------- class Animal ----------------
class Animal():
    def __init__(self, imgPath):
        self.imagePath = imgPath  # Instance variable
    def showMe(self, x, y):  # Method definition
         bg.drawImage(self.imagePath, x, y)

def pressCallback(e):
    myAnimal = Animal("sprites/animal.gif") # Object creation
    myAnimal.showMe(e.getX(), e.getY())  # Method call

makeGameGrid(600, 600, 1, False, mousePressed = pressCallback)
setBgColor(Color.green)
show()
doRun()
bg = getBg()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Les propriétés ou attributs d'un objet sont définis au moyen des variables d'instance qui prennent des valeurs distinctes pour chaque instance de la classe. Pour accéder à une variable d'instance *attribut* depuis l'une des méthodes de l'instance, on doit préfixer son nom par le fameux *self*, ce qui donne *self.attribut*. On peut aussi y accéder depuis l'extérieur de la classe en préfixant son nom d'une instance particulière comme *monInstance.attribut*.

Une classe peut également accéder aux variables et fonctions du programme principal présentes dans l'espace de noms global. Elle a ainsi par exemple accès à toutes les méthodes de la classe *GameGrid* ou à l'arrière-plan de la fenêtre de jeu référencé par la variable *bg*.

Dans le cas où l'objet ne nécessite pas d'initialisation particulière, la définition du constructeur peut être omise. Dans le programme suivant, au lieu de passer l'image de sprite au constructeur en tant qu'argument, on peut utiliser la variable *imagePath* pour se passer du constructeur.

```
from gamegrid import *
import random

# ---------------- class Animal ----------------
class Animal():
    def showMe(self, x, y):
         bg.drawImage(imagePath, x, y)

def pressCallback(e):
    myAnimal = Animal()
    myAnimal.showMe(e.getX(), e.getY())

imagePath = "sprites/animal.gif"
makeGameGrid(600, 600, 1, False, mousePressed = pressCallback)
setBgColor(Color.green)
show()
doRun()
bg = getBg()
```

HÉRITAGE, AJOUT DE MÉTHODES

Les hiérarchies de classes sont créées par dérivation de classes ou héritage, ce qui permet d'ajouter à une classe existante de nouvelles méthodes et attributs. Les instances de la classe dérivée sont également considérées des instances de la classe parente (également appelée **classe de base** ou **super classe**) et peuvent de ce fait utiliser toutes les méthodes et attributs de la classe parente. Du point de vue de la classe dérivée, c'est comme si ces méthodes ou attributs hérités étaient directement définis dans la classe dérivée elle-même.

Concrètement, un animal de compagnie possède tous les attributs et comportements d'un animal mais dispose en plus d'un nom qu'il peut afficher avec la méthode *tell()*. De ce fait, on définit une classe *Pet* (Animal de compagnie) dérivée de la classe *Animal*. Comme on veut pouvoir spécifier le nom de chaque animal de compagnie individuellement lors de sa création, on équipe le constructeur d'un paramètre formel supplémentaire *name* servant à initialiser la variable d'instance *self.name* propre à chaque animal de compagnie.


```
from gamegrid import *
from java.awt import Point

# ---------------- class Animal ----------------
class Animal():
    def __init__(self, imgPath):
        self.imagePath = imgPath
    def showMe(self, x, y):
         bg.drawImage(self.imagePath, x, y)

# ---------------- class Pet ----------------
class Pet(Animal):    # Derived from Animal
    def __init__(self, imgPath, name):
        Animal.__init__(self, imgPath)
        self.name = name
    def tell(self, x, y): # Additional method
        bg.drawText(self.name, Point(x, y))

makeGameGrid(600, 600, 1, False)
setBgColor(Color.green)
show()
doRun()
bg = getBg()
bg.setPaintColor(Color.black)

for i in range(5):
    myPet = Pet("sprites/pet.gif", "Trixi")
    myPet.showMe(50 + 100 * i, 100)
    myPet.tell(72 + 100 * i, 145)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Comme le montre le programme précédent, il est possible d'appeler la méthode *myPet.showMe() bien que celle-ci ne soit pas définie dans la classe Pet. Cela vient du fait qu'un animal de compagnie* ***est un*** *(is-a) cas particulier d'animal. La relation entretenue entre les classes Pet et Animal est de ce fait appelée une* ***relation est-un*** *(is-a en anglais).*

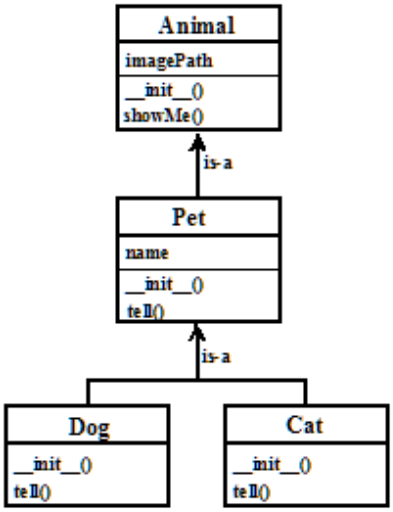
Puisque*la variable d'instance imagePath* est définie par le constructeur de la classe *Animal*, il est possible de remplacer la ligne *self.imagePath = imgPath dans le constructeur de Pet par*

Animal.__init__(self, imagePath) pour déléguer l'initialisation au constructeur de la classe de base *Animal*.

Pour définir des classes dérivées en *Python*, on spécifie le nom de la classe de base entre parenthèses juste après le nom de la classe définie. Une classe dérivée peut également hériter des méthodes et attributs de plusieurs classes de base qui seront alors mentionnées entre parenthèses et séparées par des virgules (**héritage multiple**).

HIÉRARCHIE DE CLASSES, REDÉFINITION DE MÉTHODES

Les méthodes définies dans la classe de base peuvent être redéfinies dans la classe dérivée. Il suffit d'y définir une méthode portant le même nom et avec les mêmes paramètres que dans la classe de base. Pour modéliser des chiens qui aboient avec la méthode *tell()*, on crée une classe *Dog* dérivée de *Pet* et on redéfinit la méthode *tell()* avec un comportement propre à la classe *Dog* de sorte qu'elle affiche « Waoh » (= Wouf en anglais). On peut ensuite aussi définir des chats par une classe *Cat* et leur dire de miauler en redéfinissant la méthode *tell()* de sorte qu'elle affiche « Meow » (=Miaou pour les chats anglais …).



Les quatre classes ainsi définies et permettant de représenter tous ces animaux peuvent être représentées dans un **diagramme de classes**. Celui-ci exhibe particulièrement bien la relation "est-un" entre les classes. [**plus...**].

Chaque classe est représentée par une boîte rectangulaire dans laquelle figure le nom de la classe tout en haut. Après une barre horizontale de séparation, on trouve juste sous le nom de la classe les variables d'instance. Ensuite viennent le constructeur suivi des méthodes de la classe. La hiérarchie des classes ainsi formée est facilement repérable si l'on arrange les boîtes et les flèches intelligemment.

```
from gamegrid import *

# ---------------- class Animal ----------------
class Animal():
    def __init__(self, imgPath):
        self.imagePath = imgPath
    def showMe(self, x, y):
         bg.drawImage(self.imagePath, x, y)

# ---------------- class Pet ----------------
class Pet(Animal):
    def __init__(self, imgPath, name):
        Animal.__init__(self, imgPath)
        self.name = name
    def tell(self, x, y):
        bg.drawText(self.name, Point(x, y))

# ---------------- class Dog ----------------
class Dog(Pet):
    def __init__(self, imgPath, name):
        Pet.__init__(self, imgPath, name)
    def tell(self, x, y): # Overriding
```

```
        bg.setPaintColor(Color.blue)
        bg.drawText(self.name + " tells 'Waoh'", Point(x, y))

# ---------------- class Cat ----------------
class Cat(Pet):
    def __init__(self, imgPath, name):
        Pet.__init__(self, imgPath, name)
    def tell(self, x, y): # Overriding
        bg.setPaintColor(Color.gray)
        bg.drawText(self.name + "  tells 'Meow'", Point(x, y))

makeGameGrid(600, 600, 1, False)
setBgColor(Color.green)
show()
doRun()
bg = getBg()

alex = Dog("sprites/dog.gif", "Alex")
alex.showMe(100, 100)
alex.tell(200, 130)  # Overriden method is called

rex = Dog("sprites/dog.gif", "Rex")
rex.showMe(100, 300)
rex.tell(200, 330)  # Overriden method is called

xara = Cat("sprites/cat.gif", "Xara")
xara.showMe(100, 500)
xara.tell(200, 530)  # Overriden method is called
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

On peut modifier le comportement de la classe de base en la dérivant et en redéfinissant ses méthodes dans la classe dérivée. Lors de l'invocation de méthodes de la même classe ou d'une classe de base, il ne faut pas oublier le *self* qui doit impérativement préfixer leur nom. Attention cependant à ne pas insérer le *self* en tant que premier argument lors de l'invocation d'une méthode. Python s'en charge à la place du programmeur.

Parfois, il faut pouvoir invoquer depuis la méthode *XX()* surchargée dans la classe dérivée *ChildClass* la méthode *XX()* originale définie dans la classe parente *BaseClass*. Pour réaliser cela, il faut utiliser la syntaxe *BaseClass.XX(self, …)* en prenant bien soin de transmettre ensuite les paramètres nécessaires [**plus...**].

Cette règle s'applique très souvent au constructeur *__init__()* de la classe dérivée si l'on veut invoquer le constructeur de la classe de base depuis le constructeur de la classe dérivée. Au sein de la classe dérivée, l'appel au constructeur de la classe de base ressemblera donc à

```
class BaseClass:

    def __init__(self, a):
        self.a = a

class ChildClass(BaseClass):

    def __init__(self, a, b):
        # on délègue au constructeur de la classe de base l'initialisation de
        # la variable d'instance `a`
        BaseClass.__init__(self, a)
        self.b = b
```

POLYMORPHISME: APPEL DE MÉTHODES DIRIGÉE PAR LE TYPE

Le polymorphisme est un peu plus ardu à comprendre mais il constitue une fonctionnalité essentielle de la programmation orientée objets. Il concerne l'appel des fonctions redéfinies (overridden methods) dans les classes dérivées en adaptant l'appel en fonction de la classe à laquelle l'instance est affiliée. Voici un exemple qui permettra de clarifier cette notion : un programme définit une liste *Animals* contenant des instances d'animaux de compagnie de types différents

```
animals = [Dog(), Dog(), Cat()]
```

Le fait de parcourir cette liste et d'invoquer la méthode *tell()* poserait problème sans le polymorphisme car il y a trois méthodes différentes nommées *tell()* parmi les classes *Pet*, *Dog* et *Cat*.

```
for animal in animals:
    animal.tell()
```

L'interpréteur Python pourrait résoudre cette ambiguïté de trois manières différentes :

1. En déclenchant une erreur
2. En appelant la méthode *tell()* de la classe de base *Pet*
3. En invoquant la version de *tell()* appropriée, selon le type de chaque animal

Dans un langage polymorphique tel que Python, c'est la troisième solution, de loin la meilleure, qui est utilisée.

```
from gamegrid import *
from soundsystem import *

# ---------------- class Animal ----------------
class Animal():
    def __init__(self, imgPath):
        self.imagePath = imgPath
    def showMe(self, x, y):
         bg.drawImage(self.imagePath, x, y)

# ---------------- class Pet ----------------
class Pet(Animal):
    def __init__(self, imgPath, name):
        Animal.__init__(self, imgPath)
        self.name = name
    def tell(self, x, y):
        bg.drawText(self.name, Point(x, y))

# ---------------- class Dog ----------------
class Dog(Pet):
    def __init__(self, imgPath, name):
         Pet.__init__(self, imgPath, name)
    def tell(self, x, y): # Overridden
         Pet.tell(self, x, y)
         openSoundPlayer("wav/dog.wav")
         play()

# ---------------- class Cat ----------------
class Cat(Pet):
    def __init__(self, imgPath, name):
        Pet.__init__(self, imgPath, name)
    def tell(self, x, y): # Overridden
        Pet.tell(self, x, y)
        openSoundPlayer("wav/cat.wav")
        play()

makeGameGrid(600, 600, 1, False)
setBgColor(Color.green)
show()
```

```
doRun()
bg = getBg()

animals = [Dog("sprites/dog.gif", "Alex"),
     Dog("sprites/dog.gif", "Rex"),
     Cat("sprites/cat.gif", "Xara")]

y = 100
for animal in animals:
    animal.showMe(100, y)
    animal.tell(200, y + 30)     # Which tell()????
    show()
    y = y + 200
    delay(1000)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Le polymorphisme fait en sorte que ce soit l'affiliation à la classe qui détermine laquelle des méthodes sera invoquée dans le cas de méthodes redéfinies dans les classes filles. Du fait qu'en Python l'appartenance à une certaine classe n'est déterminée qu'au moment de l'exécution, le polymorphisme est trivial.

Le typage dynamique utilisé en Python est appelée **duck test** ou **duck typing** selon la citation attribuée à James Whitcomb Riley (1849 – 1916) :

« When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck. »

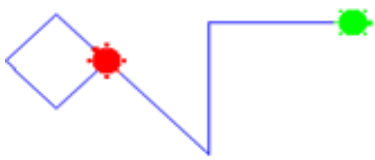
Lorsque je vois un oiseau (classe de base *Bird*) qui marche comme un canard, nage comme un canard et cancane comme un canard (méthodes redéfinies dans la classe spécialisée *Duck*), je considère cet oiseau comme étant un canard (Classe dérivée *Duck*).

Il arrive qu'une méthode redéfinie soit définie dans la classe de base et que son corps soit vide. Pour cela, on peut se limiter à y mettre une instruction **return** (qui ne fait donc que de retourner *None*) ou l'expression vide **pass.**

EXERCICES

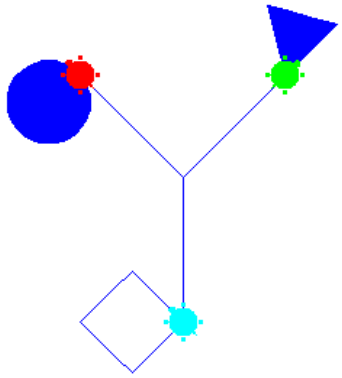
1. Définir une classe *TurtleKid* dérivée de la classe *Turtle* dont la méthode *shape()* dessine un carré. Une fois cette classe dérivée définie, le code suivant devrait fonctionner sans erreur:

```
tf = TurtleFrame()
# john is a Turtle
john = Turtle(tf)
# john knows all commands of Turtle
john.setColor("green")
john.forward(100)
john.right(90)
john.forward(100)

# laura is a TurtleKid, but also a Turtle
# laura knows all commands of Turtle
laura = TurtleKid(tf)
laura.setColor("red")
laura.left(45)
laura.forward(100)
# laura knows a new command too
laura.shape()
```



2. Définir deux classes dérivées de *Turtle*, à savoir *TurtleBoy* et *TurtleGirl* qui redéfinissent la méthode *shape()* de sorte qu'une instance *TurtleBoy* dessine un triangle plein et qu'une instance *TurtleGirl* dessine un disque plein. Le programme suivant devrait alors pouvoir s'exécuter sans problème:

```
tf = TurtleFrame()

aGirl = TurtleGirl(tf)
aGirl.setColor("red")
aGirl.left(45)
aGirl.forward(100)
aGirl.shape()

aBoy = TurtleBoy(tf)
aBoy.setColor("green")
aBoy.right(45)
aBoy.forward(100)
aBoy.shape()

aKid = TurtleKid(tf)
aKid.back(100)
aKid.left(45)
aKid.shape()
```



3. Dessiner le diagramme de classes de l'exercice 2.

MATÉRIEL SUPPLÉMENTAIRE

VARIABLES ET MÉTHODES STATIQUES

Les classes peuvent également être utilisées pour regrouper des variables et fonctions qui ont un but commun dans le but de rendre le code plus lisible. On pourrait par exemple regrouper les principales constantes physiques dans la classe *Physics*. Des variables définies directement sous l'en-tête de la classe sont appelées **variables statiques** et on s'y réfère en préfixant leur nom de celui de la classe dont elles font partie. Au contraire des variables d'instance, il n'est pas nécessaire de créer une instance de la classe pour y accéder.

```
import math

# ---------------- class Physics ----------------
class Physics():
    # Avogadro constant [mol-1]
    N_AVOGADRO = 6.0221419947e23
    # Boltzmann constant [J K-1]
    K_BOLTZMANN = 1.380650324e-23
    # Planck constant [J s]
    H_PLANCK = 6.6260687652e-34;
    # Speed of light in vacuo [m s-1]
    C_LIGHT = 2.99792458e8
    # Molar gas constant [K-1 mol-1]
    R_GAS = 8.31447215
    # Faraday constant [C mol-1]
    F_FARADAY = 9.6485341539e4;
    # Absolute zero [Celsius]
    T_ABS = -273.15
    # Charge on the electron [C]
    Q_ELECTRON = -1.60217646263e-19
    # Electrical permittivity of free space [F m-1]
    EPSILON_0 = 8.854187817e-12
    # Magnetic permeability of free space [ 4p10-7 H m-1 (N A-2)]
    MU_0 = math.pi*4.0e-7
```

```
c = 1 / math.sqrt(Physics.EPSILON_0 * Physics.MU_0)
print("Speed of light (calulated): %s m/s" %c)
print("Speed of light (table): %s  m/s" %Physics.C_LIGHT)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

On peut également regrouper des fonctions apparentées en les déclarant comme statiques dans une même classe. On peut ensuite utiliser ces méthodes en préfixant leur nom par celui de leur classe sans avoir à créer une instance de cette classe .

Pour déclarer une méthode comme statique, il faut écrire la ligne *@staticmethod* juste au-dessus de sa définition.

```
# ---------------- class OhmsLaw ----------------
class OhmsLaw():
    @staticmethod
    def U(R, I):
        return R * I

    @staticmethod
    def I(U, R):
        return U / R

    @staticmethod
    def R(U, I):
        return U / I

r = 10
i = 1.5

u = OhmsLaw.U(r, i)
print("Voltage = %s V" %u)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Les variables statiques (également appelées **variables de classe**) appartiennent à la classe elle-même alors que les variables d'instance sont rattachées à une instance particulière de la classe. Ainsi, toutes les instances d'une même classe partagent les variables de classe entre elles alors que chacune dispose de ses propres copies des variables d'instance. On peut lire et modifier une variable de classe en préfixant son nom de celui de sa classe suivi d'un point.

On utilise typiquement une variable de classe pour implémenter un **compteur d'instances** générées à partir d'une classe données. On crée alors une variable de classe *counter* et on incrémente cette variable de classe dans le constructeur *__init__* à la création de chaque instance.

Des fonctions apparentées peuvent être regroupées en tant que méthodes de classe (statiques) dans une classe au nom bien choisi. La ligne *@staticmethod* est un **décorateur de fonctions** et doit figurer juste au-dessus de la fonction ou méthode décorée.

7.3 JEUX D'ARCADE, FROGGER

INTRODUCTION

De nombreux jeux vidéo rencontrés sur les consoles de jeux ou sur Internet sont constitués d'images qui se déplacent sur une image d'arrière-fond. Il arrive même que l'arrière-fond soit en mouvement, particulièrement lorsque les personnages du jeu se déplacent vers les bords de la fenêtre de jeu. Cet effet procure au joueur la sensation que le monde du jeu est bien plus grand que l'aire de jeu visible. Bien que les images d'animations demandent beaucoup de ressources de calcul, les concepts sous-jacents n'en sont pas pour autant difficiles à comprendre et à assimiler. On peut dire pour l'instant que la boucle de jeu (*game loop* en anglais) recalcule le contenu de l'écran, copie l'image d'arrière-fond ainsi que les sprites des personnages dans une mémoire tampon invisible (*buffer* en anglais) de manière à pouvoir tout afficher en une seule fois dans la fenêtre. Dès que l'on dépasse les 25 images par seconde dans les animations (25 fps = Frame Per Second), les mouvements seront assez fluides tandis qu'en-dessous, les animations paraîtront saccadées.

Dans de nombreux jeux, les personnages interagissent entre eux par des collisions, ce qui fait de la détection et de la gestion de collisions un élément central d'un moteur de jeu. Une bibliothèque de jeux développée de manière minutieuse telle que *JGameGrid* met à disposition des programmeurs des mécanismes de détection de collisions contrôlables à l'aide de la programmation événementielle. Le programmeur se charge de définir quels objets peuvent rentrer en collision et le système détecte par lui-même lorsqu'il y a collision et lance la fonction de rappel que le programmeur lui aura fourni.

CONCEPTS DE PROGRAMMATION: *Conception de jeux, sprite, acteur /personnage, collision, superviseur*

SCÉNARIO DE JEU

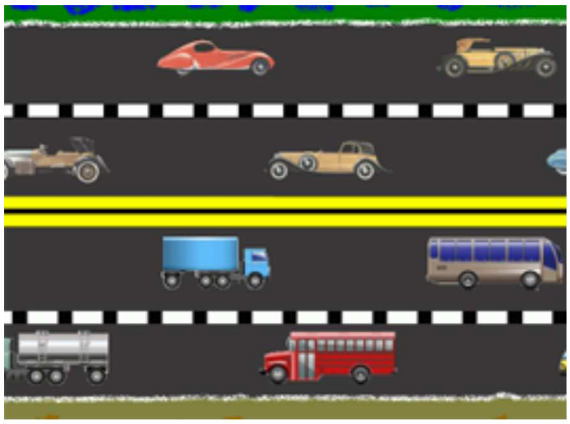
Lors du développement d'un jeu vidéo, il est très important de commencer par mettre au point un scénario aussi détaillé que possible permettant de rédiger des spécifications fonctionnelles du programme. Très souvent, on a tendance à viser trop haut pour la première mouture du jeu au lieu de commencer simplement, en implémentant les fonctionnalités de base et en améliorant le jeu par versions successives. La grande idée est de développer de manière à ce que le programme soit facilement extensible par la suite, sans qu'il soit nécessaire de modifier des tas de lignes de code dans tous les coins pour supporter l'extension voulue. L'idéal est qu'il soit possible d'ajouter des nouvelles fonctionnalités en n'ayant pratiquement rien à retoucher au code déjà existant. Dans la pratique, même les meilleurs développeurs ont parfois bien de la peine à prévoir toutes les éventualités et à écrire un code parfaitement évolutif. Voilà pourquoi le développement de jeu est souvent fait de grands moments de joie et de satisfaction mais aussi de moments très frustrants. Cela ne fera qu'amplifier votre plaisir et votre satisfaction de pouvoir finalement présenter votre propre jeu à vos amis.

En attendant de devenir un programmeur chevronné, il est recommandé de développer des clones de jeux bien connus en y insérant votre petite touche personnelle à l'aide d'images de sprites personnalisées. Durant cette phase de formation, il n'est pas très important que ce développement de jeux aboutisse à des produits parfaits car il ne s'agit pas avant tout d'y jouer des heures mais surtout d'apprendre comment ils sont développés. Un jeu très connu s'appelle « Frogger » et possède le scénario suivant :

Une grenouille tente de traverser une route noire de trafic pour parvenir au marécage

situé de l'autre côté. Si la tortue rentre en collision avec un véhicule, la grenouille perd une vie. Le but est d'utiliser les touches directionnelles du clavier pour amener la grenouille saine et sauve au marécage.

Il faut implémenter quatre voies de circulation : deux voies empruntées par des camions et des bus roulant en sens inverse et deux autres voies fréquentées par des voitures anciennes roulant en sens inverse (voir l'image ci-contre).



On commence par développer les mouvements de la grenouille et des véhicules. Ensuite, il faudra ajouter la gestion des collisions, le calcul des points ainsi que les conditions de fin de jeu (game over).

Dans la *GameGrid*, les véhicules sont modélisés comme des instances de la classe *Car* qui dérive elle-même de *Actor*. Les mouvements des véhicules sont programmés dans la méthode **act()**.

On utilisera les images *car0.gif, …, car19.gif présentes dans la distribution de TigerJython. Il est bien entendu possible d'utiliser des images personnalisées qui ont pour dimensions au maximum* 70 pixels de hauteur et 200 pixels de largeur avec un fond transparent.

Dans les jeux d'arcade, on utilise habituellement un plateau de jeu dont les cases de la grille ont une largeur de un pixel, de sorte que la grille du jeu coïncide avec la grille de pixels de l'écran. On fixe la taille de la fenêtre à 800 x 600 pixels et on affiche en image de fond la route contenue dans le fichier *lane.gif* dont les dimensions sont de 801 x 601 pixels. Il nous faut encore générer 20 véhicules avec la fonction **initCars()** et décider de leur position et angle de visée initiaux sur le plateau de jeu.

Il est facile de déplacer les véhicules dans la méthode **act()** en les décalant à l'aide de la méthode *move()*. Lorsqu'un véhicule roulant de gauche à droite sort de l'écran à droite, on le fait réapparaître à gauche et lorsqu'un véhicule roulant en sens inverse disparaît à gauche de l'écran, on le fait réapparaître à droite. Rappelez-vous qu'un acteur de jeu (en l'occurrence les véhicules) peut posséder des coordonnées correspondant à une position hors écran.

```
from gamegrid import *

# ---------------- class Car ----------------
class Car(Actor):
    def __init__(self, path):
        Actor.__init__(self, path)

    def act(self):
        self.move()
        if self.getX() < -100:
            self.setX(1650)
        if self.getX() > 1650:
            self.setX(-100)

def initCars():
    for i in range(20):
        car = Car("sprites/car" + str(i) + ".gif")
        if i < 5:
            addActor(car, Location(350 * i, 100), 0)
        if i >= 5 and i < 10:
            addActor(car, Location(350 * (i - 5), 220), 180)
        if i >= 10 and i < 15:
            addActor(car, Location(350 * (i - 10), 350), 0)
        if i >= 15:
            addActor(car, Location(350 * (i - 15), 470), 180)
```

```
makeGameGrid(800, 600, 1, None, "sprites/lane.gif", False)
setSimulationPeriod(50)
initCars()
show()
doRun()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Pour les jeux d'arcade, on utilise habituellement une grille de jeu dont les cellules font 1 pixel de largeur. On effectue le rendu de la scène du jeu 20 fois par seconde en fixant la période de simulation à 50 ms, ce qui permet un mouvement relativement fluide. Des saccades sporadiques peuvent survenir en raison d'un manque de puissance du CPU. En effet, du fait que le rendu graphique s'effectue sans accélération graphique et que le CPU est limité au niveau des traitements graphiques, la période de simulation peut difficilement être abaissée en-dessous de 50 ms.

DÉPLACER LA GRENOUILLE AVEC LES FLÈCHES DIRECTIONNELLES

On peut maintenant songer à incorporer la grenouille dans le jeu. Elle doit apparaître au bas de l'écran et se mouvoir à l'aide des flèches directionnelles du clavier.

Comme elle est également un acteur, il faut commencer par écrire la classe *Frog* dérivant de la classe *Actor*. Il suffit d'y définir le constructeur *__init__* sans se soucier de la méthode *move()* qui est inutile puisque les déplacements de la grenouille sont commandés par les événements clavier déclenchés par le joueur. On définit le gestionnaire d'événements clavier par la fonction de rappel **onKeyRepeated**,

que l'on enregistre à l'aide du paramètre nommé *keyRepeated* lors de la création de la grille de jeu avec *makeGameGrid()*. Ce gestionnaire d'événement *onKeyRepeated* sera non seulement appelé lors de la pression d'une touche du clavier, mais également, de manière répétée, lorsque la touche sera maintenue enfoncée.

On détermine la touche enfoncée au sein du gestionnaire d'événement *onKeyRepeated* et on déplace la grenouille de 5 cellules (5 pixels) dans la direction correspondante.

```
from gamegrid import *

# ---------------- class Frog ----------------
class Frog(Actor):
  def __init__(self):
      Actor.__init__(self, "sprites/frog.gif")

# ---------------- class Car ----------------
class Car(Actor):
    def __init__(self, path):
        Actor.__init__(self, path)

    def act(self):
        self.move()
        if self.getX() < -100:
            self.setX(1650)
        if self.getX() > 1650:
            self.setX(-100)

def initCars():
    for i in range(20):
        car = Car("sprites/car" + str(i) + ".gif")
        if i < 5:
            addActor(car, Location(350 * i, 100), 0)
```

```
        if i >= 5 and i < 10:
            addActor(car, Location(350 * (i - 5), 220), 180)
        if i >= 10 and i < 15:
            addActor(car, Location(350 * (i - 10), 350), 0)
        if i >= 15:
            addActor(car, Location(350 * (i - 15), 470), 180)

def onKeyRepeated(keyCode):
    if keyCode == 37: # left
        frog.setX(frog.getX() - 5)
    elif keyCode == 38: # up
        frog.setY(frog.getY() - 5)
    elif keyCode == 39: # right
        frog.setX(frog.getX() + 5)
    elif keyCode == 40: # down
        frog.setY(frog.getY() + 5)


makeGameGrid(800, 600, 1, None, "sprites/lane.gif", False,
             keyRepeated = onKeyRepeated)
setSimulationPeriod(50);
frog = Frog()
addActor(frog, Location(400, 560), 90)
initCars()
show()
doRun()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

On pourrait également capturer les événements clavier à l'aide des fonctions de rappel *keyPressed(e)* et *keyReleased(e)*. À la différence de *keyRepeated(code)*, il faudrait alors récupérer le code de touche à partir du paramètre événement *e* par l'appel *e.getKeyCode()*. De plus, les événements *keyPressed(e)* et *keyReleased(e)* sont moins adaptés dans ce jeu car ils causent un délai entre la première pression de la touche et le moment où la répétition des événements de pression est déclenchée.

Pour connaître le code des touches directionnelles, il faut concocter un petit programme de test qui les affiche sur la sortie standard :

```
from gamegrid import *

def onKeyPressed(e):
    print "Pressed: ", e.getKeyCode()

def onKeyReleased(e):
    print "Released: ", e.getKeyCode()

makeGameGrid(800, 600, 1, None, "sprites/lane.gif", False,
  keyPressed = onKeyPressed, keyReleased = onKeyReleased)
show()
```

ÉVÉNEMENTS DE COLLISION

Il est relativement simple de détecter les collisions entre acteurs. Il suffit de spécifier, lors de la création d'un véhicule *car*, que la grenouille doit déclencher un événement particulier lors d'une collision avec ce véhicule, à l'aide de l'appel

```
frog.addCollisionActor(car)
```

Ceci va faire que chaque collision déclenche automatiquement la méthode **collide()** de la classe *Frog*. Il suffit de traiter l'événement de collision de manière appropriée dans cette méthode

collide(), en faisant par exemple sauter la grenouille à sa position initiale au bas de l'écran.

```
from gamegrid import *

# ---------------- class Frog ----------------
class Frog(Actor):
    def __init__(self):
        Actor.__init__(self, "sprites/frog.gif")
        self.setCollisionCircle(Point(0, -10), 5)

    def collide(self, actor1, actor2):
        self.setLocation(Location(400, 560))
        return 0

# ---------------- class Car ----------------
class Car(Actor):
    def __init__(self, path):
        Actor.__init__(self, path)

    def act(self):
        self.move()
        if self.getX() < -100:
            self.setX(1650)
        if self.getX() > 1650:
            self.setX(-100)

def initCars():
    for i in range(20):
        car = Car("sprites/car" + str(i) + ".gif")
        frog.addCollisionActor(car)
        if i < 5:
            addActor(car, Location(350 * i, 100), 0)
        if i >= 5 and i < 10:
            addActor(car, Location(350 * (i - 5), 220), 180)
        if i >= 10 and i < 15:
            addActor(car, Location(350 * (i - 10), 350), 0)
        if i >= 15:
            addActor(car, Location(350 * (i - 15), 470), 180)

def onKeyRepeated(keyCode):
    if keyCode == 37: # left
        frog.setX(frog.getX() - 5)
    elif keyCode == 38: # up
        frog.setY(frog.getY() - 5)
    elif keyCode == 39: # right
        frog.setX(frog.getX() + 5)
    elif keyCode == 40: # down
        frog.setY(frog.getY() + 5)

makeGameGrid(800, 600, 1, None, "sprites/lane.gif", False,
     keyRepeated = onKeyRepeated)
setSimulationPeriod(50)
frog = Frog()
addActor(frog, Location(400, 560), 90)
initCars()
show()
doRun()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

La méthode **collide()** n'est en l'occurrence pas une fonction de rappel mais une méthode de la classe *Actor* redéfinie dans la classe *Frog*. C'est la raison pour laquelle il n'est pas nécessaire d'enregistrer la méthode *collide* en tant que fonction de rappel avec un paramètre nommé.

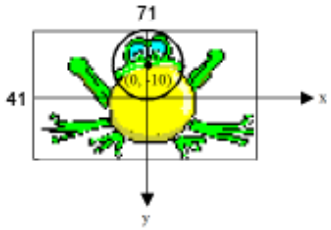
Par défaut, un événement collision est déclenché lorsque les rectangles circonscrits aux sprites se recoupent. Il est cependant possible de modifier la forme, la taille et la position de la zone de détection de collision pour s'adapter à l'image du sprite. A cet effet, on peut utiliser les méthodes suivantes de la classe *Actor* :

Methode	**Collision area**
setCollisionCircle(centerPoint, radius)	Cercle de centre et de rayon donnés (en pixels)
setCollisionImage()	Pixels non transparents du sprite associé à l'acteur. Ne fonctionne qu'avec un partenaire de collision disposant d'une zone de collision en cercle, ligne ou point
setCollisionLine(startPoint, endPoint)	Segment droit défini par les points *start* et *end*
setCollisionRectangle(center, width, height)	Rectangle de centre *center*, de largeur *width* et de hauteur *height*
setCollisionSpot(spotPoint)	Point dont les coordonnées sont fixées de manière relative au centre du sprite associé à l'acteur

Toutes les méthodes utilisent un système de coordonnées en pixels relatif dont l'origine se trouve au centre du sprite. L'axe *Ox* est orienté vers la droite et l'axe *Oy* est orienté vers le bas.

La taille de l'image de la grenouille est de 71 x 41 pixels. Ainsi, il est par exemple possible d'ajouter les lignes suivantes au constructeur de la classe *Frog*

```
self.setCollisionCircle(Point(0, -10), 5)
```



afin de définir une zone de collision circulaire de 5 pixels de rayon autour de la tête de la grenouille. Un véhicule doit donc entrer dans ce cercle pour déclencher une collision avec la grenouille.
Comme les collisions sont mises en mémoire cache pour des raisons de performance, il peut être nécessaire de redémarrer TigerJython pour que les changements au niveau des zones de détection soient prises en compte.

GESTIONNAIRE DE JEU ET SON

Dans le domaine des jeux de société, il est souvent nécessaire de désigner une personne comme « maître du jeu » responsable de veiller au respect des règles, de distribuer les points et de désigner le vainqueur à l'issue de la partie. De manière similaire, il est également plus judicieux de ne pas attribuer à un acteur en particulier le soin de gérer le déroulement du jeu mais de programmer cet aspect dans une partie indépendante du programme. La partie principale du programme est bien adaptée à cet effet puisque son exécution se poursuit après l'initialisation du jeu. Il suffit d'ajouter à la fin du programme principal une boucle qui va périodiquement tester l'état du jeu et y réagir de manière appropriée. Il faut cependant veiller à insérer un délai avec **delay()** entre chaque itération de la boucle pour éviter de surcharger inutilement le processeur qui ne pourrait plus s'occuper des autres parties du jeu comme le mouvement des acteurs. Cette boucle devrait se terminer lors de la fermeture de la fenêtre de jeu, ce qui a comme consequence que **isDisposed()** retourne *True*. Le gestionnaire de jeu peut par exemple limiter le nombre de vies de la grenouille ou compter et afficher le nombre de traversées réussies et ratées.

Il est souvent assez délicat de gérer les conditions de fin de jeu car il faut considérer de nombreux cas différents. On veut aussi souvent pouvoir jouer plusieurs parties de suite après un « game over » sans devoir redémarrer tout le jeu.

N'hésitez pas à utiliser les connaissances acquises dans le chapitre *Son* pour insérer des effets sonores et ainsi améliorer le « game play ». Le plus simple est d'utiliser la fonction *playTone()*.

```
from gamegrid import *

# ---------------- class Frog ----------------
class Frog(Actor):
    def __init__(self):
        Actor.__init__(self, "sprites/frog.gif")

    def collide(self, actor1, actor2):
        global nbHit
        nbHit += 1
        playTone([("c''h'a'f'", 100)])
        self.setLocation(Location(400, 560))
        return 0

    def act(self):
        global nbSuccess
        if self.getY() < 15:
            nbSuccess += 1
            playTone([("c'e'g'c''", 200)])
            self.setLocation(Location(400, 560))

# ---------------- class Car ----------------
class Car(Actor):
    def __init__(self, path):
        Actor.__init__(self, path)

    def act(self):
        self.move()
        if self.getX() < -100:
            self.setX(1650)
        if self.getX() > 1650:
            self.setX(-100)

def initCars():
    for i in range(20):
        car = Car("sprites/car" + str(i) + ".gif")
        frog.addCollisionActor(car)
        if i < 5:
            addActor(car, Location(350 * i, 90), 0)
        if i >= 5 and i < 10:
            addActor(car, Location(350 * (i - 5), 220), 180)
        if i >= 10 and i < 15:
            addActor(car, Location(350 * (i - 10), 350), 0)
        if i >= 15:
            addActor(car, Location(350 * (i - 15), 470), 180)

def onKeyRepeated(keyCode):
    if keyCode == 37: # left
        frog.setX(frog.getX() - 5)
    elif keyCode == 38: # up
        frog.setY(frog.getY() - 5)
    elif keyCode == 39: # right
        frog.setX(frog.getX() + 5)
    elif keyCode == 40: # down
        frog.setY(frog.getY() + 5)

makeGameGrid(800, 600, 1, None, "sprites/lane.gif", False,
    keyRepeated = onKeyRepeated)
setSimulationPeriod(50)
setTitle("Frogger")
frog = Frog()
addActor(frog, Location(400, 560), 90)
initCars()
show()
doRun()
```

```
# Game supervision
maxNbLifes = 3
nbHit = 0
nbSuccess = 0
while not isDisposed():
    if nbHit + nbSuccess == maxNbLifes:  # game over
        addActor(Actor("sprites/gameover.gif"), Location(400, 285))
        removeActor(frog)
        doPause()
    setTitle("#Success: " + str(nbSuccess) + " #Hits " + str(nbHit))
    delay(100)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Le comptage des traversées réussies avec **nbSuccess** et des échecs avec **nbHit** s'effectue dans la classe *Frog*. Ceci explique pourquoi ces variables doivent être déclarées comme globales.

Lors de la fin du jeu, une image portant l'inscription « Game Over » est insérée, la grenouille est supprimée, le cycle de simulation est stoppé avec **doPause()** et, finalement, la boucle du gestionnaire de jeu est interrompue avec *break*. On aurait aussi pu utiliser un acteur textuel de la classe *TextActor pour afficher « Game Over », ce qui permettrait de changer le texte lors de* l'exécution.

```
rate = nbSuccess / (nbSuccess + nbHit)
ta = TextActor("  Game Over: Success Rate = " + str(rate) + " % ",
     DARKGRAY, YELLOW, Font("Arial", Font.BOLD, 24))
addActor(ta, Location(200, 287))
```

EXERCICES

1. Remplacer l'image de fond et les vieilles voitures avec des images d'animaux qui nagent dans une rivière (Crocodiles, etc.).

2. Introduire un système de gestion des points et un temps limite pour effectuer la traversée : chaque traversée réussie rapporte 5 points, chaque collision fait perdre 5 points. Dépasser la limite de temps ramène la grenouille à sa position d'origine et fait perdre 10 points.

3. Étendre le jeu en faisant tourner votre imagination et en implémentant quelques nouveautés sympathiques.

7.4 GRILLE DE JEU, JEU DE PLATEAU, SOLITAIRE

INTRODUCTION

Dans certains jeux vidéo, le positionnement des figurines de jeu est restreint aux cases d'une structure en grille qui sont de même taille et arrangées sous forme de matrice. Tenir compte de cette restriction à une structure de grille simplifie beaucoup l'implémentation du jeu. Comme son nom l'indique, la bibliothèque *JGameGrid* est partifculièrement optimisée pour les jeux sous forme de grille.

Dans ce chapitre, nous allons développer par étapes le jeu du solitaire avec la disposition de plateau anglaise. Nous verrons également différentes méthodes applicables à tous les jeux sous forme de grille.

CONCEPTS DE PROGRAMMATION: *Plateau de jeu, règles du jeu, spécifications, fin de jeu*

INITIALISATION DU PLATEAU, CONTRÔLE AVEC LA SOURIS

Le plateau est formé d'un arrangement régulier de trous dans lesquels sont disposées des billes en marbre ou, dans certaines variantes, des bâtonnets. Le plateau de solitaire le plus connu, le plateau anglais, comporte 33 trous disposés en forme de croix. Au départ, tous les trous sont occupés par une bille sauf le trou central. Comme son nom l'indique, ce jeu est généralement joué en solo. Les règles sont les suivantes : un mouvement consiste à choisir intelligemment une bille puis la déplacer jusque dans un trou libre en sautant par-dessus une autre bille adjacente, soit horizontalement, soit verticalement. On supprime alors du plateau la bille par-dessus laquelle on a sauté.

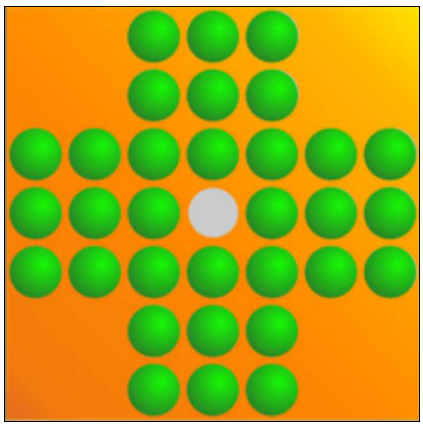
Un plateau anglais de *Solitaire* d'Inde, 1830

Le but est de se débarrasser de toutes les billes du plateau excepté la dernière. Si la dernière bille se trouve au milieu du plateau en fin de jeu, on considère que la résolution du jeu est particulièrement élégante. Dans son implémentation sur ordinateur, il faudrait pouvoir prendre une bille indiquée par un clic de souris pour la déplacer par glisser-déposer. Lorsque le bouton est relâché, le jeu doit vérifier que le mouvement effectué respecte les règles du jeu. Si les règles sont violées, la bille doit retourner à sa place initiale et, dans le cas contraire, il faut faire apparaître une bille au nouvel emplacement et supprimer celle se trouvant dans le trou initial.

La spécification du jeu est ainsi clarifiée et la phase d'implémentation peut débuter. Comme d'habitude, on procède par étapes en s'assurant que le jeu s'exécute convenablement à chaque modification du code. Il est parfaitement naturel d'utiliser une grille de jeu de 7x7 cellules sans pour autant utiliser celles qui se trouvent dans les coins. On commence par dessiner la grille avec la fonction *initBoard()* en utilisant comme arrière-fond l'image *solitaire_board.png* disponible dans la distribution *TigerJython*. On implémente le contrôle à l'aide de la souris en utilisant les gestionnaires d'événements *mousePressed*, *mouseDragged*, et *mouseReleased*.

Lors d'un événement **Press event**, on garde trace de la cellule courante et de la bille qui s'y trouve. On peut obtenir cette bille à l'aide de **getOneActorAt()** qui retourne *None* si la cellule est vide. Écrire les informations importantes dans la barre d'état de la fenêtre de jeu ou dans la console permet de faciliter le processus de développement et de résoudre plus facilement les erreurs.

Durant la phase de **glisser de la souris** on déplace l'image de la bille à la position actuelle du curseur de la souris à l'aide de **setLocationOffset()**. On évite de restreindre le mouvement aux positions centrales des cellules pour véritablement suivre le curseur et donner lieu à un mouvement continu. Pour éviter les problèmes de superposition d'acteurs dans la grille, il est important de ne déplacer que l'image de sprite de l'acteur et non l'acteur-bille en lui-même qui reste dans sa case jusqu'à ce que le mouvement soit validé (d'où le nom de la fonction : *Offset* = décalage). Dans cette première version du jeu, la bille déplacée doit simplement revenir à sa position initiale après l'événement **Release event** lors du relâchement de la souris. Ceci se réalise avec l'appel **setLocationOffset(0, 0).**



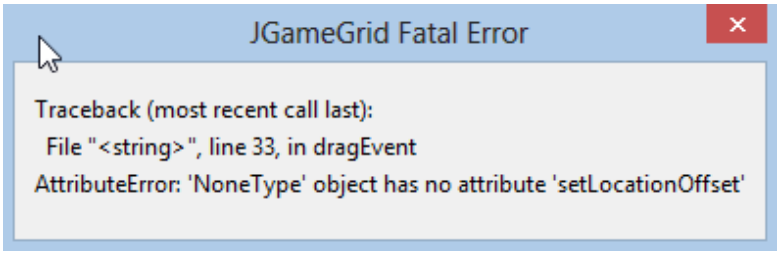
```
from gamegrid import *

def isMarbleLocation(loc):
    if loc.x < 0 or loc.x > 6 or loc.y < 0 or loc.y > 6:
        return False
    if loc.x in [0, 1, 5, 6] and loc.y in [0, 1, 5, 6]:
        return False
    return True

def initBoard():
    for x in range(7):
        for y in range(7):
            loc = Location(x, y)
            if isMarbleLocation(loc):
                marble = Actor("sprites/marble.png")
                addActor(marble, loc)
    removeActorsAt(Location(3, 3)) # Remove marble in center

def pressEvent(e):
    global startLoc, movingMarble
    startLoc = toLocationInGrid(e.getX(), e.getY())
    movingMarble = getOneActorAt(startLoc)
    if movingMarble == None:
       setStatusText("Pressed at " + str(startLoc) + ". No marble found")
    else:
       setStatusText("Pressed at " + str(startLoc) + ". Marble found")

def dragEvent(e):
    if movingMarble == None:
        return
    startPoint = toPoint(startLoc)
    movingMarble.setLocationOffset(e.getX() - startPoint.x,
                                   e.getY() - startPoint.y)

def releaseEvent(e):
    if movingMarble == None:
        return
    movingMarble.setLocationOffset(0, 0)

makeGameGrid(7, 7, 70, None, "sprites/solitaire_board.png", False,
    mousePressed = pressEvent, mouseDragged = dragEvent,
    mouseReleased = releaseEvent)
setBgColor(Color(255, 166, 0))
setSimulationPeriod(20)
addStatusBar(30)
setStatusText("Press-drag-release to make a move.")
initBoard()
show()
doRun()
```

MEMENTO

Au lieu de réellement déplacer un acteur lors du glisser-déposer, il est souvent préférable de déplacer uniquement son image de sprite à l'aide de **setLocationOffset(x, y)** par rapport au central de l'acteur.

Il est très important de faire une distinction claire entre les coordonnées de la souris et les coordonnées des cellules au sein de la grille lors de la gestion des mouvements de la souris. On utilisera les fonctions *toLocationInGrid(pixel_coord)* et *toPoint(location_coord)* pour passer d'un système de coordonnées à l'autre.

Si l'on tente de déplacer un acteur depuis une cellule vide, les événements de glisser-déposer vont conduire à un crash du programme causé par un appel à *movingMarble.setLocationOffset* alors que *movingMarble* vaut *None* et ne représente par conséquent pas un objet possédant la méthode *setLocationOffset*.



Pour éviter ce malheureux message d'erreur, on place au début des gestionnaires d'événements un test qui termine la fonction avec *return* si aucune bille n'est sélectionnée.

IMPLÉMENTER LES RÈGLES DU JEU

Comment vérifier que les règles du jeu sont bien respectées ? Il faut certainement avoir connaissance de la bille qui est en train d'être déplacée par l'intermédiaire de ses coordonnées *start* dans la grille de jeu. Il faut également connaître les coordonnées de destination *dest*. Les conditions suivantes doivent être réalisées pour que le mouvement soit valide:

1. La cellule *start contient* une bille
2. La cellule *dest* ne contient pas de bille
3. *dest* est une cellule appartenant au plateau du jeu
4. *start* et *dest* ne sont séparées que par une seule cellule qui leur est adjacente à toutes deux horizontalement ou verticalement
5. Il y a une bille dans la cellule située entre *start* et *end*

Il est judicieux d'implémenter ces conditions dans une fonction **getRemoveMarble(start, dest)** qui retourne la bille qui doit être supprimée après un mouvement autorisé et qui retourne *None* en cas de mouvement illégal.

De ce fait, il faudrait appeler cette fonction lors du relâchement de la souris et utiliser **removeActor()** pour supprimer du plateau l'acteur retourné si le mouvement est légal.

```
from gamegrid import *

def getRemoveMarble(start, dest):
    if getOneActorAt(start) == None:
        return None
    if getOneActorAt(dest) != None:
        return None
    if not isMarbleLocation(dest):
        return None
    if dest.x - start.x == 2 and dest.y == start.y:
        return getOneActorAt(Location(start.x + 1, start.y))
```

```
    if start.x - dest.x == 2 and dest.y == start.y:
        return getOneActorAt(Location(start.x - 1, start.y))
    if dest.y - start.y == 2 and dest.x == start.x:
        return getOneActorAt(Location(start.x, start.y + 1))
    if start.y - dest.y == 2 and dest.x == start.x:
        return getOneActorAt(Location(start.x, start.y - 1))

def isMarbleLocation(loc):
    if loc.x < 0 or loc.x > 6 or loc.y < 0 or loc.y > 6:
        return False
    if loc.x in [0, 1, 5, 6] and loc.y in [0, 1, 5, 6]:
        return False
    return True

def initBoard():
    for x in range(7):
        for y in range(7):
            loc = Location(x, y)
            if isMarbleLocation(loc):
                marble = Actor("sprites/marble.png")
                addActor(marble, loc)
    removeActorsAt(Location(3, 3)) # Remove marble in center

def pressEvent(e):
    global startLoc, movingMarble
    startLoc = toLocationInGrid(e.getX(), e.getY())
    movingMarble = getOneActorAt(startLoc)
    if movingMarble == None:
       setStatusText("Pressed at " + str(startLoc) + ". No marble found")
    else:
       setStatusText("Pressed at " + str(startLoc) + ". Marble found")

def dragEvent(e):
    if movingMarble == None:
        return
    startPoint = toPoint(startLoc)
    movingMarble.setLocationOffset(e.getX() - startPoint.x,
                                   e.getY() - startPoint.y)

def releaseEvent(e):
    if movingMarble == None:
        return
    destLoc = toLocationInGrid(e.getX(), e.getY())
    movingMarble.setLocationOffset(0, 0)
    removeMarble = getRemoveMarble(startLoc, destLoc)
    if removeMarble == None:
        setStatusText("Released at " + str(destLoc) + ". Not a valid move.")
    else:
        removeActor(removeMarble)
        movingMarble.setLocation(destLoc)
        setStatusText("Released at " + str(destLoc)+ ". Marble removed.")


startLoc = None
movingMarble = None

makeGameGrid(7, 7, 70, None, "sprites/solitaire_board.png", False,
    mousePressed = pressEvent, mouseDragged = dragEvent,
    mouseReleased = releaseEvent)
setBgColor(Color(255, 166, 0))
setSimulationPeriod(20)
addStatusBar(30)
setStatusText("Press-drag-release to make a move.")
initBoard()
show()
doRun()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

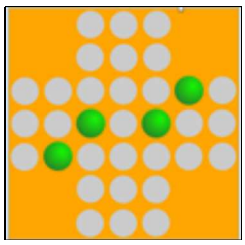
MEMENTO

Au lieu d'utiliser plusieurs tests *if* et instructions *return* consécutifs pour quitter la fonction *getRemoveMarble()*, il serait également possible d'utiliser une combinaison de toutes ces conditions à l'aide d'un opérateur booléen. Aucune des alternatives n'est objectivement meilleure que l'autre : il s'agit plutôt d'une question de goût.

TESTER LES CONDITIONS DE FIN DE JEU

À présent, il ne reste plus qu'à tester les conditions de fin de jeu à l'issue de chaque mouvement. Le jeu est certainement terminé s'il ne reste plus qu'une seule boule sur le plateau et, dans ce cas, le but du jeu est atteint.

Il ne faut cependant pas oublier les nombreuses configurations possibles dans lesquelles le jeu est considéré comme terminé même si le but n'est pas atteint. Cette situation survient lorsqu'il reste plus d'une boule sur le plateau mais qu'il n'est plus possible d'effectuer un mouvement conforme aux règles. On ne sait pas *a priori* s'il est possible de tomber dans cette situation en ne faisant que des mouvements légaux mais il faut toujours programmer **de manière défensive** pour se prémunir contre l'imprévu. En programmation, il faut toujours croire à la loi de Murphy : « Lorsqu'il y a un truc qui peut foirer, ça va foirer ».



Pour se tirer d'affaire, on peut définir une fonction **isMovePossible()**. qui teste pour chaque bille restante si elle peut être utilisée dans un mouvement légal. *isMovePossible()* va tester pour chaque bille s'il existe une bille adjacente par-dessus laquelle elle peut sauter pour atterrir dans un trou vide juste au-delà [**plus...**].

```
from gamegrid import *

def checkGameOver():
    global isGameOver
    marbles = getActors() # get remaining marbles
    if len(marbles) == 1:
        setStatusText("Game over. You won.")
        isGameOver = True
    else:
        # check if there are any valid moves left
        if not isMovePossible():
           setStatusText("Game over. You lost. (No valid moves available)")
           isGameOver = True

def isMovePossible():
   for a in getActors():  # run over all remaining marbles
        for x in range(7): # run over all holes
            for y in range(7):
                loc = Location(x, y)
                if getOneActorAt(loc) == None and \
                  getRemoveMarble(a.getLocation(), Location(x, y)) != None:
                    return True
   return False

def getRemoveMarble(start, dest):
    if getOneActorAt(start) == None:
        return None
    if getOneActorAt(dest) != None:
        return None
    if not isMarbleLocation(dest):
        return None
    if dest.x - start.x == 2 and dest.y == start.y:
        return getOneActorAt(Location(start.x + 1, start.y))
    if start.x - dest.x == 2 and dest.y == start.y:
```

```
        return getOneActorAt(Location(start.x - 1, start.y))
    if dest.y - start.y == 2 and dest.x == start.x:
        return getOneActorAt(Location(start.x, start.y + 1))
    if start.y - dest.y == 2 and dest.x == start.x:
        return getOneActorAt(Location(start.x, start.y - 1))
    return None

def isMarbleLocation(loc):
    if loc.x < 0 or loc.x > 6 or loc.y < 0 or loc.y > 6:
        return False
    if loc.x in [0, 1, 5, 6] and loc.y in [0, 1, 5, 6]:
        return False
    return True

def initBoard():
    for x in range(7):
        for y in range(7):
            loc = Location(x, y)
            if isMarbleLocation(loc):
                marble = Actor("sprites/marble.png")
                addActor(marble, loc)
    removeActorsAt(Location(3, 3)) # Remove marble in center

def pressEvent(e):
    global startLoc, movingMarble
    if isGameOver:
        return
    startLoc = toLocationInGrid(e.getX(), e.getY())
    movingMarble = getOneActorAt(startLoc)
    if movingMarble == None:
       setStatusText("Pressed at " + str(startLoc) + ".No marble found")
    else:
       setStatusText("Pressed at " + str(startLoc) + ".Marble found")

def dragEvent(e):
    if isGameOver:
        return
    if movingMarble == None:
        return
    startPoint = toPoint(startLoc)
    movingMarble.setLocationOffset(e.getX() - startPoint.x,
                                   e.getY() - startPoint.y)

def releaseEvent(e):
    if isGameOver:
        return
    if movingMarble == None:
        return
    destLoc = toLocationInGrid(e.getX(), e.getY())
    movingMarble.setLocationOffset(0, 0)
    removeMarble = getRemoveMarble(startLoc, destLoc)
    if removeMarble == None:
        setStatusText("Released at " + str(destLoc)
                      + ". Not a valid move.")
    else:
        removeActor(removeMarble)
        movingMarble.setLocation(destLoc)
        setStatusText("Released at " + str(destLoc)+
                      ". Valid move - Marble removed.")
        checkGameOver()


startLoc = None
movingMarble = None
isGameOver = False

makeGameGrid(7, 7, 70, None, "sprites/solitaire_board.png", False,
   mousePressed = pressEvent, mouseDragged = dragEvent,
   mouseReleased = releaseEvent)
```

```
setBgColor(Color(255, 166, 0))
setSimulationPeriod(20)
addStatusBar(30)
setStatusText("Press-drag-release to make a move.")
initBoard()
show()
doRun()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

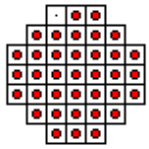
MEMENTO

Après chaque mouvement, il faut vérifier si le jeu est terminé avec **checkGameOver()** et, le cas échéant, ajuster le fanion booléen **isGameOver = True** qui indique en tout temps l'état du jeu.

En particulier, il ne faut pas oublier d'empêcher tout déplacement des billes par la souris avec un **return** conditionnel placé tout au début de chaque gestionnaire d'événements concernant la souris.

EXERCICES

1. Créer un plateau de solitaire français.



2. Étendre le solitaire avec un score qui compte et afficher le nombre de mouvements effectués Il faudrait également pouvoir redémarrer le jeu à l'aide de la touche espace du clavier.

3. Familiarisez-vous avec les stratégies de résolution du solitaire grâce aux conseils d'un connaisseur ou des nombreuses ressources disponibles sur le Web [**plus...**]

4. Créer un plateau de solitaire d'après votre propre imagination.

7.5 SPRITES ANIMÉS

INTRODUCTION

Lorsque l'on travaille avec la bibliothèque *JGameGrid*, tous les acteurs devraient être dérivés de la classe *Actor* pour qu'ils réutilisent automatiquement de nombreuses fonctionnalités et capacités incluses cette dernière sans aucun effort de programmation. Leur apparence propre doit cependant être chargée depuis un fichier image spécifique appelé **sprite**.

Les personnages de jeu sont animés de différentes manières : ils bougent à travers la surface de jeu en changeant parfois d'apparence, de posture ou d'expression. De ce fait, on peut attribuer à un objet acteur de nombreux sprites différents que l'on distingue par un indice entier, l'ID du sprite. Cette technique est plus simple que de spécialiser les différentes apparences sous forme de classes dérivées. Les personnages de jeu modifient donc souvent leur position, direction de mouvement et angle de rotation. L'orientation du sprite devrait être automatiquement ajustée à la direction de son mouvement. Avec *JGameGrid*, il faut spécifier dès la création de l'acteur, pour des raisons de performance, si ce dernier peut subir des rotations et, le cas échéant, quels sont les sprites correspondant aux différentes orientations possibles. Ces derniers sont, dès l'instanciation de la classe *Actor*, chargés dans une mémoire tampon qui contient également les différentes rotations des images. Ceci permet d'éviter de devoir, lors de l'exécution du jeu, charger les images depuis le disque dur ou de leur appliquer des transformations, ce qui dégraderait les performances. Par défaut, pour chaque sprite chargé depuis une image, 60 images de sprites sont générées pour des angles de rotations entre 0° et 360°, par incréments de 6°. *JGameGrid* fait usage d'un concept d'animation également disponible dans d'autres bibliothèques de développement de jeu, notamment **Greenfoot** [**plus...**] .

Principe fondamental de l'animation:

> La méthode *act(),* telle que définie dans la classe *Actor(),* possède un corps vide et retourne donc immédiatement. Pour définir des acteurs personnalisés, il faut donc créer des classes dérivées de *Actor* et y redéfinir la méthode *act() pour implémenter leur comportement propre.*
>
> Lors de l'ajout d'un acteur à la fenêtre de jeu avec *addActor(),* celui-ci va être inséré dans une liste *actOrder,* ordonnée selon la classe de l'acteur. Une boucle de jeu interne, en l'occurrence appelée cycle de simulation, va parcourir périodiquement cette liste et appeler de manière séquentielle la méthode *act()* propre à chaque acteur grâce au mécanisme de polymorphisme d'héritage.

Pour que ce principe astucieux fonctionne correctement, il est nécessaire que les acteurs adoptent un **comportement coopératif** : leur méthode *act()* doit se contenter d'exécuter un code qui **retourne très rapidement**. L'insertion de boucles ou de délais dans la méthode *act()*, ne serait-ce que d'un seul acteur, entraîne des effets catastrophiques puisque les autres acteurs devront ainsi attendre les uns sur les autres pour voir leur méthode *act()* être traitée.

Il faut avoir conscience du mécanisme qui régit le rendu des images de sprite des acteurs présents dans le jeu. Dans la boucle de jeu, les sprites de chaque acteur sont copiés dans une mémoire tampon regroupant les images de tous les acteurs selon l'ordre spécifié par la liste **paintOrder**. C'est à partir de cette mémoire tampon que le rendu des acteurs est effectué dans la fenêtre de jeu. L'ordre de rendu des acteurs détermine donc leur visibilité : le sprite des derniers acteurs dessinés va se situer au premier plan tandis que celui des premiers acteurs sera en arrière-plan. Du fait que les acteurs sont insérés dans la liste **paintOrder** selon l'ordre d'ajout avec *addActor()*, les acteurs ajoutés en dernier apparaîtront au premier plan. Heureusement,

autant l'ordre de la liste **actOrder** que celui de **paintOrder** peuvent être modifiés lors de l'exécution. En particulier, un acteur peut demander à être placé en tête de liste avec *setOnTop()* pour que son sprite apparaisse au premier plan et que sa méthode *act()* soit invoquée en priorité

Bien qu'il soit possible d'assigner un nombre arbitraire de sprites à un acteur lors de sa création, il n'est ensuite plus possible de les modifier lors de l'exécution. Si un acteur (par exemple un texte de légende) doit changer d'apparence dynamiquement lors de l'exécution du jeu, il est possible de générer l'acteur dynamiquement grâce aux fonctions graphiques habituelles.

CONCEPTS DE PROGRAMMATION: *Cycle de simulation, code coopératif, fabrique de classe, variable statique (de classe), découplage*

BOUGER UNE ARBALÈTE ET UNE FLÈCHE

Développons un jeu dans lequel le joueur bouge, à l'aide des flèches directionnelles, une arbalète (crossbow) qui tire des flèches (arrows) empruntant une trajectoire naturelle (parabole). Les flèches devront atteindre des fruits volant pour les pourfendre.

On définit une classe *Crossbow* qui dérive de la classe *Actor*. Pour indiquer qu'il s'agit d'un acteur pouvant subir des rotations, on indique *True* lors de l'appel du constructeur de la classe de base. Le troisième paramètre, en l'occurrence l'entier 2, indique qu'il y a deux images de sprite associées avec l'arbalète, l'une représentant l'arbalète tendue et chargée d'une flèche et l'autre représentant l'arbalète au repos et dépourvue de flèche. Les images correspondantes sont automatiquement cherchées sous le nom *sprites/crossbow_0.gif* et *sprites/crossbow_1.gif* dans la distribution de *TigerJython*.

```
Actor.__init__(self, True, "sprites/crossbow.gif", 2)
```

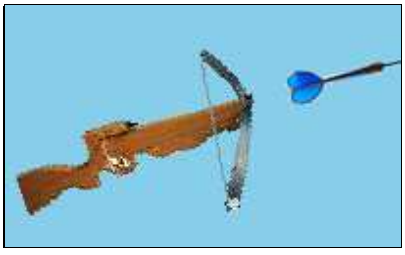
L'arbalète est contrôlée par des événements clavier : on change son orientation avec les touches haut/bas et on la déclenche avec la touche espace. La fonction de rappel *keyCallback()* qui sert de gestionnaire d'événements est enregistrée lors de l'appel à *makeGameGrid() avec le paramètre nommé keyPressed*.

La classe *Dart* modélisant les flèches est un peu compliquée du fait que ces dernières doivent se mouvoir sur une trajectoire parabolique dans un système de coordonnées *xOy* dont l'axe *Ox* est horizontal vers la droite et l'axe *Oy* vertical et orienté vers le bas. La trajectoire n'est pas déterminée par une équation de courbe mais plutôt parcourue itérativement par de petits changements temporels *dt*. On sait de la cinématique que les nouvelles composantes du vecteur vitesse (*vx', vy'*) et du vecteur position (*px', py'*) sont déterminées après chaque incrément temporel *dt* comme suit, où *g = 9.81m/s^2* est l'accélération de la pesanteur à la surface de la terre :

$$vx' = vx$$
$$vy' = vy + g * dt$$

$$px' = px + vx * dt$$
$$py' = py + vy * dt$$

On détermine les valeurs initiales dans la méthode **reset()** qui est appelée automatiquement lors de l'ajout d'une instance de *Dart* à la fenêtre de jeu.



On peut donner à la flèche une nouvelle position et orientation dans sa méthode *act()*. Pour ne pas gaspiller de ressources de calcul, il faut supprimer la flèche du jeu sitôt qu'elle sort de la fenêtre et la replacer sur l'arbalète en position de tir.

```
from gamegrid import *
```

```
import math

# ------------------- class Crossbow ----------------------
class Crossbow(Actor):
    def __init__(self):
        Actor.__init__(self, True, "sprites/crossbow.gif", 2)

# ------ class Dart ----------------
class Dart(Actor):
    def __init__(self, speed):
        Actor.__init__(self, True, "sprites/dart.gif")
        self.speed = speed
        self.dt = 0.005 * getSimulationPeriod()

    # Called when actor is added to GameGrid
    def reset(self):
        self.px = self.getX()
        self.py = self.getY()
        self.vx = self.speed * math.cos(math.radians(self.getDirection()))
        self.vy = self.speed * math.sin(math.radians(self.getDirection()))

    def act(self):
        self.vy = self.vy + g * self.dt
        self.px = self.px + self.vx * self.dt
        self.py = self.py + self.vy * self.dt
        self.setLocation(Location(int(self.px), int(self.py)))
        self.setDirection(math.degrees(math.atan2(self.vy, self.vx)))
        if not self.isInGrid():
            self.removeSelf()
            crossbow.show(0) # Load crossbow

# ------ End of class definitions --------------------

def keyCallback(e):
    code = e.getKeyCode()
    if code == KeyEvent.VK_UP:
        crossbow.setDirection(crossbow.getDirection() - 5)
    elif code == KeyEvent.VK_DOWN:
        crossbow.setDirection(crossbow.getDirection() + 5)
    elif code == KeyEvent.VK_SPACE:
        if crossbow.getIdVisible() == 1: # Wait until crossbow is loaded
            return
        crossbow.show(1) # crossbow is released
        dart = Dart(100)
        addActorNoRefresh(dart, crossbow.getLocation(),
                                crossbow.getDirection())

screenWidth = 600
screenHeight = 400
g = 9.81

makeGameGrid(screenWidth, screenHeight, 1, False, keyPressed = keyCallback)
setTitle("Use Cursor up/down to target, Space to shoot.")
setBgColor(makeColor("skyblue"))
crossbow = Crossbow()
addActor(crossbow, Location(80, 320))
setSimulationPeriod(30)
doRun()
show()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Lors de l'appel du constructeur de la classe *Actor*, on peut indiquer si l'acteur va subir des rotations et s'il faut lui associer plusieurs images de sprite. Les images de sprite sont chargées à ce moment depuis le disque et placées dans une mémoire tampon qui contiendra également

toutes les différentes rotations des images.

On règle sans cesse l'orientation de la flèche pour qu'elle corresponde à la direction de son vecteur vitesse dans le but de rendre son vol plus naturel.

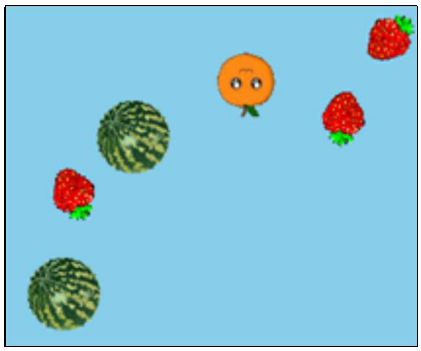
FABRIQUE DE FRUITS ET FRUITS EN MOUVEMENT

Notre programme va utiliser trois différents types de fruits : des melons, des oranges et des fraises. Ceux-ci sont générés en continu de manière aléatoire et se déplacent depuis le coin supérieur droit de la fenêtre vers la gauche avec une vitesse horizontale initiale aléatoire et suivant une trajectoire parabolique. Ces trois différents types de fruits partagent de nombreuses caractéristiques communes et ne comportent que quelques différences mineures. Il ne serait de ce fait pas une bonne idée de dériver ces trois classes directement de la classe de base *Actor* car cela impliquerait de coder les méthodes qui leur seraient communes dans chacune des classes, ce qui constituerait une **redondance de code** à éviter à tout prix. Dans cette situation, il est plus judicieux de définir une classe auxiliaire et intermédiaire *Fruit* dans laquelle sont implémentées les fonctionnalités communes et de laquelle les classes spécifiques *Melon*, *Orange* et *Strawberry* peuvent être dérivées.

On délègue la création d'un fruit à la classe *FruitFactory* d'un genre particulier : il s'agit d'une **classe fabrique**. Bien que celle-ci ne possède pas d'image de sprite, on peut néanmoins la dériver de la classe *Actor* de telle sorte que sa méthode *act()* puisse être utilisée pour générer de nouveaux fruits. Cette classe fabrique a une particularité : bien qu'elle produise de nombreux fruits, il n'y aura qu'une seule instance de cette classe dans tout le jeu [**plus...**]. De ce fait, on ne définit généralement pas de constructeur pour une telle classe puisqu'elle n'est pas destinée à engendrer plusieurs instances. On définit plutôt une méthode nommée **create()** ou de manière similaire qui crée une unique instance de la classe et la retourne comme valeur de retour. Tous les appels subséquents à cette méthode **create()**, au lieu de recréer une nouvelle instance, ne vont faire que de renvoyer la référence à l'instance précédemment créée.

Comme la méthode **create()** est invoquée sans passer par une instance particulière, il s'agit d'une méthode de classe qui doit de ce fait être décorée avec *@staticmethod*.

Lors de la création de la *FruitFactory*, le nombre maximum de fruits que la fabrique peut créer est spécifié dans une variable *capacity*. De plus, chaque *Actor* peut invoquer la méthode *setSlowDown()* pour ralentir la fréquence d'appels à la méthode *act()*.



```
from gamegrid import *
import random

# ---------- class Fruit -------------------------
class Fruit(Actor):
    def __init__(self, spriteImg, vx):
        Actor.__init__(self, True, spriteImg, 2) # rotatable, 2 sprites
        self.vx = vx
        self.vy = 0

    def reset(self): # Called when Fruit is added to GameGrid
        self.px = self.getX()
        self.py = self.getY()

    def act(self):
        self.movePhysically()
        self.turn(10)

    def movePhysically(self):
        self.dt = 0.002 * getSimulationPeriod()
```

```
        self.vy = self.vy + g * self.dt # vx = const
        self.px = self.px + self.vx * self.dt
        self.py = self.py + self.vy * self.dt
        self.setLocation(Location(int(self.px), int(self.py)))
        self.cleanUp()

    def cleanUp(self):
        if not self.isInGrid():
            self.removeSelf()

# ------ class Melon -----------
class Melon(Fruit):
    def __init__(self, vx):
        Fruit.__init__(self, "sprites/melon.gif", vx)

# ------ class Orange -----------
class Orange(Fruit):
    def __init__(self, vx):
        Fruit.__init__(self, "sprites/orange.gif", vx)

# ------ class Strawberry -----------
class Strawberry(Fruit):
    def __init__(self, vx):
        Fruit.__init__(self, "sprites/strawberry.gif", vx)

# ------------------ class FruitFactory -------------------
class FruitFactory(Actor):
    myFruitFactory = None
    myCapacity = 0
    nbGenerated = 0

    @staticmethod
    def create(capacity, slowDown):
        if FruitFactory.myFruitFactory == None:
            FruitFactory.myCapacity = capacity
            FruitFactory.myFruitFactory = FruitFactory()
            FruitFactory.myFruitFactory.setSlowDown(slowDown)
                # slows down act() call for this actor
        return FruitFactory.myFruitFactory

    def act(self):
        if FruitFactory.nbGenerated == FruitFactory.myCapacity:
            print "Factory expired"
            return

        vx = -(random.random() * 20 + 30)
        r = random.randint(0, 2)
        if r == 0:
            fruit = Melon(vx)
        elif r == 1:
            fruit = Orange(vx)
        else:
            fruit = Strawberry(vx)
        FruitFactory.nbGenerated += 1
        y = int(random.random() * screenHeight / 2)
        addActorNoRefresh(fruit, Location(screenWidth-50, y), 180)

# ------ End of class definitions --------------------

FACTORY_CAPACITY = 20
FACTORY_SLOWDOWN = 35
screenWidth = 600
screenHeight = 400
g = 9.81

makeGameGrid(screenWidth, screenHeight, 1, False)
setTitle("Use Cursor up/down to target, Space to shoot.")
setBgColor(makeColor("skyblue"))
factory = FruitFactory.create(FACTORY_CAPACITY, FACTORY_SLOWDOWN)
```

```
addActor(factory, Location(0, 0))  # needed to run act()
setSimulationPeriod(30)
doRun()
show()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Dans une méthode statique, il n'y a pas de paramètre *self*. De ce fait, les variables créées au sein de la méthode *create()* doivent être des **variables statiques** (leur nom doit être préfixé du nom de la classe) [**plus...**].

Lors de l'ajout d'acteurs dans la grille de jeu avec *addActor()*, les images présentes dans la mémoire tampon sont automatiquement affichées à l'écran de sorte que l'acteur est immédiatement visible. Dès que le cycle de simulation est démarré, le rendu est de toute manière effectué à chaque cycle de simulation. C'est la raison pour laquelle, dans le cas présent, il faut utiliser **addActorNoRefresh()** pour éviter qu'un rendu trop fréquent n'entraine des scintillements à l'écran.

GESTION ET INTÉGRATION DES COLLISIONS

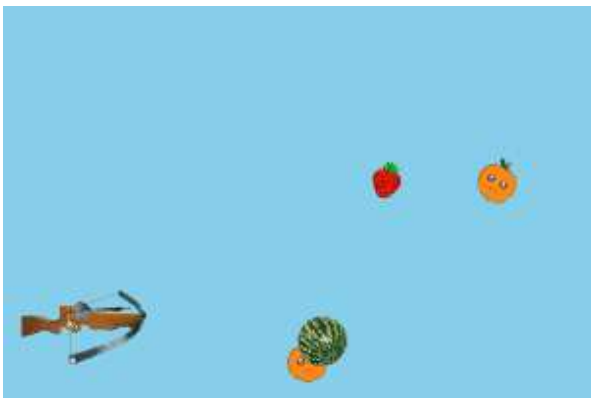
Les deux parties du programme que nous venons de mettre en place auraient pu être développées indépendamment par deux équipes de développeurs distincts. Si le **style de programmation est cohérent** et en majorité **découplé** à l'exemple de notre code, il est presque un jeu d'enfant de fusionner les deux parties.

Nous allons donc mettre ces deux pièces ensemble et ajouter au passage une nouvelle fonctionnalité permettant aux fruits d'être fendus lorsqu'ils sont atteints par la flèche. Le terrain est déjà préparé puisque les fruits disposent de deux images de sprite : l'une pour le fruit entier et l'autre pour le fruit fendu.

Comme vous le savez, les collisions entre acteurs sont détectées par des événements de collision et leur gestion nécessite d'identifier, pour chaque acteur, les partenaires de collision potentiels. En ce qui concerne la flèche, il s'agit de tous les fruits existants et visibles à l'écran.

Il ne faut cependant pas oublier que davantage de fruits sont ajoutés à la grille de jeu pendant le vol de la flèche. Ceci exige de déclarer également toutes les flèches (même s'il n'y en a probablement qu'une seule) comme partenaires de collision lors de la création d'un fruit.

Avec JGameGrid, il est également possible de passer toute une liste d'acteurs à la fonction **addCollisionActors()** en tant que partenaires de collision. La fonction *getActors(class)* permet d'obtenir une liste de tous les acteurs appartenant à la classe mentionnée que l'on peut passer à *addCollisionActors()*.



```
from gamegrid import *
import random
import math

# ---------- class Fruit ------------------------
class Fruit(Actor):
    def  __init__(self, spriteImg, vx):
        Actor.__init__(self, True, spriteImg, 2)
        self.vx = vx
        self.vy = 0
        self.isSliced = False

    def reset(self): # Called when Fruit is added to GameGrid
        self.px = self.getX()
        self.py = self.getY()

    def act(self):
        self.movePhysically()
        self.turn(10)

    def movePhysically(self):
        self.dt = 0.002 * getSimulationPeriod()
        self.vy = self.vy + g * self.dt
        self.px = self.px + self.vx * self.dt
        self.py = self.py + self.vy * self.dt
        self.setLocation(Location(int(self.px), int(self.py)))
        self.cleanUp()

    def cleanUp(self):
        if not self.isInGrid():
            self.removeSelf()

    def sliceFruit(self):
        if not self.isSliced:
            self.isSliced = True
            self.show(1)

    def collide(self, actor1, actor2):
       actor1.sliceFruit()
       return 0

# ------ class Melon -----------
class Melon(Fruit):
    def __init__(self, vx):
        Fruit.__init__(self, "sprites/melon.gif", vx)

# ------ class Orange -----------
class Orange(Fruit):
    def __init__(self, vx):
        Fruit.__init__(self, "sprites/orange.gif", vx)

# ------ class Strawberry -----------
class Strawberry(Fruit):
    def __init__(self, vx):
        Fruit.__init__(self, "sprites/strawberry.gif", vx)

# ------------------- class FruitFactory -------------------
class FruitFactory(Actor):
    myCapacity = 0
    myFruitFactory = None
    nbGenerated = 0

    @staticmethod
    def create(capacity, slowDown):
        if FruitFactory.myFruitFactory == None:
            FruitFactory.myCapacity = capacity
            FruitFactory.myFruitFactory = FruitFactory()
            FruitFactory.myFruitFactory.setSlowDown(slowDown)
        return FruitFactory.myFruitFactory
```

```
    def act(self):
        self.createRandomFruit()

    def createRandomFruit(self):
        if FruitFactory.nbGenerated == FruitFactory.myCapacity:
            print "Factory expired"
            return

        vx = -(random.random() * 20 + 30)
        r = random.randint(0, 2)
        if r == 0:
            fruit = Melon(vx)
        elif r == 1:
            fruit = Orange(vx)
        else:
            fruit = Strawberry(vx)
        FruitFactory.nbGenerated += 1
        y = int(random.random() * screenHeight / 2)
        addActorNoRefresh(fruit, Location(screenWidth-50, y), 180)
        # for a new fruit, the collision partners are all existing darts
        fruit.addCollisionActors(toArrayList(getActors(Dart)))

# ------------------- class Crossbow ----------------------
class Crossbow(Actor):
    def __init__(self):
        Actor.__init__(self, True, "sprites/crossbow.gif", 2)

# ------ class Dart ----------------
class Dart(Actor):
    def __init__(self, speed):
        Actor.__init__(self, True, "sprites/dart.gif")
        self.speed = speed
        self.dt = 0.005 * getSimulationPeriod()

    # Called when actor is added to GameGrid
    def reset(self):
        self.px = self.getX()
        self.py = self.getY()
        dx = math.cos(math.radians(self.getDirectionStart()))
        self.vx = self.speed * dx
        dy = math.sin(math.radians(self.getDirectionStart()))
        self.vy = self.speed * dy

    def act(self):
        self.vy = self.vy + g * self.dt
        self.px = self.px + self.vx * self.dt
        self.py = self.py + self.vy * self.dt
        self.setLocation(Location(int(self.px), int(self.py)))
        self.setDirection(math.degrees(math.atan2(self.vy, self.vx)))
        if not self.isInGrid():
            self.removeSelf()
            crossbow.show(0) # Load crossbow

    def collide(self, actor1, actor2):
        actor2.sliceFruit()
        return 0

# ------ End of class definitions -------------------

def keyCallback(e):
    code = e.getKeyCode()
    if code == KeyEvent.VK_UP:
        crossbow.setDirection(crossbow.getDirection() - 5)
    elif code == KeyEvent.VK_DOWN:
        crossbow.setDirection(crossbow.getDirection() + 5)
    elif code == KeyEvent.VK_SPACE:
        if crossbow.getIdVisible() == 1: # Wait until crossbow is loaded
            return
```

```
        crossbow.show(1) # crossbow is released
        dart = Dart(100)
        addActorNoRefresh(dart, crossbow.getLocation(),
                                crossbow.getDirection())
        # for a new dart, the collision partners are all existing fruits
        dart.addCollisionActors(toArrayList(getActors(Fruit)))


FACTORY_CAPACITY = 20
FACTORY_SLOWDOWN = 35
screenWidth = 600
screenHeight = 400
g = 9.81

makeGameGrid(screenWidth, screenHeight, 1, False, keyPressed = keyCallback)
setTitle("Use Cursor up/down to target, Space to shoot.")
setBgColor(makeColor("skyblue"))
factory = FruitFactory.create(FACTORY_CAPACITY, FACTORY_SLOWDOWN)
addActor(factory, Location(0, 0))  # needed to run act()
crossbow = Crossbow()
addActor(crossbow, Location(80, 320))
setSimulationPeriod(30)
doRun()
show()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Une fois que les partenaires de collision d'un acteur sont déclarés, il faut redéfinir la méthode *collide()* dans la classe de cet acteur pour qu'elle soit appelée lors de chaque collision. La valeur de retour doit être un nombre entier indiquant pendant combien de cycles de simulation, la détection de collision doit être suspendue (en l'occurrence 0). Un nombre supérieur à 0 est parfois nécessaire pour que les deux partenaires aient le temps de se séparer avant qu'une nouvelle collision soit détectée.

Notez bien que dans la définition de la méthode **collide(self, actor1, actor2)**, *actor1* est l'acteur de la classe dans laquelle *collide()* est définie.

Par défaut, la zone de détection de collisions correspond au rectangle circonscrit à l'image du sprite. Ce rectangle est évidemment tourné avec l'image lors des rotations. On pourrait définir la zone de collision des flèches comme un cercle de faible rayon autour de leur pointe, pour éviter que les fruits ne soient fendus lors d'une collision avec la queue d'une flèche.

```
setCollisionCircle(Point(20, 0), 10)
```

AFFICHAGE DE L'ÉTAT DU JEU ET GESTION DE LA FIN DU JEU

Pour le dessert, raffinons encore notre jeu en y ajoutant des informations à destination de l'utilisateur ainsi que le décompte des points. Le plus simple est de les écrire dans la barre d'état de la fenêtre de jeu.

Comme nous l'avons déjà vu, il est judicieux d'implémenter les règles du jeu dans un gestionnaire de jeu indépendant des acteurs dans le programme principal. Ce dernier affiche le nombre de fruits touchés, le nombre de ceux qui ont été manqués et termine le jeu lorsque la fabrique de fruits est épuisée. Le gestionnaire affiche également le score final, génère un acteur textuel *Game Over* et empêche le jeu de se poursuivre.

```
from gamegrid import *
import random
import math
```

```
# ---------- class Fruit ------------------------
class Fruit(Actor):
    def  __init__(self, spriteImg, vx):
        Actor.__init__(self, True, spriteImg, 2)
        self.vx = vx
        self.vy = 0
        self.isSliced = False

    def reset(self): # Called when Fruit is added to GameGrid
        self.px = self.getX()
        self.py = self.getY()

    def act(self):
        self.movePhysically()
        self.turn(10)

    def movePhysically(self):
        self.dt = 0.002 * getSimulationPeriod()
        self.vy = self.vy + g * self.dt
        self.px = self.px + self.vx * self.dt
        self.py = self.py + self.vy * self.dt
        self.setLocation(Location(int(self.px), int(self.py)))
        self.cleanUp()

    def cleanUp(self):
        if not self.isInGrid():
            if not self.isSliced:
                FruitFactory.nbMissed += 1
            self.removeSelf()

    def sliceFruit(self):
        if not self.isSliced:
            self.isSliced = True
            self.show(1)
            FruitFactory.nbHit += 1

    def collide(self, actor1, actor2):
       actor1.sliceFruit()
       return 0

# ------ class Melon -----------
class Melon(Fruit):
    def __init__(self, vx):
        Fruit.__init__(self, "sprites/melon.gif", vx)

# ------ class Orange -----------
class Orange(Fruit):
    def __init__(self, vx):
        Fruit.__init__(self, "sprites/orange.gif", vx)

# ------ class Strawberry -----------
class Strawberry(Fruit):
    def __init__(self, vx):
        Fruit.__init__(self, "sprites/strawberry.gif", vx)

# ------------------- class FruitFactory -------------------
class FruitFactory(Actor):
    myCapacity = 0
    myFruitFactory = None
    nbGenerated = 0
    nbMissed = 0
    nbHit = 0

    @staticmethod
    def create(capacity, slowDown):
        if FruitFactory.myFruitFactory == None:
            FruitFactory.myCapacity = capacity
            FruitFactory.myFruitFactory = FruitFactory()
            FruitFactory.myFruitFactory.setSlowDown(slowDown)
```

```
        return FruitFactory.myFruitFactory

    def act(self):
        self.createRandomFruit()

    @staticmethod
    def createRandomFruit():
        if FruitFactory.nbGenerated == FruitFactory.myCapacity:
            return
        vx = -(random.random() * 20 + 30)
        fruitClass = random.choice([Melon, Orange, Strawberry])
        fruit = fruitClass(vx)
        FruitFactory.nbGenerated += 1
        y = int(random.random() * screenHeight / 2)
        addActorNoRefresh(fruit, Location(screenWidth-50, y), 180)
        # for a new fruit, the collision partners are all existing darts
        fruit.addCollisionActors(toArrayList(getActors(Dart)))

# ------------------ class Crossbow ----------------------
class Crossbow(Actor):
    def __init__(self):
        Actor.__init__(self, True, "sprites/crossbow.gif", 2)

# ------ class Dart ----------------
class Dart(Actor):
    def __init__(self, speed):
        Actor.__init__(self, True, "sprites/dart.gif")
        self.speed = speed
        self.dt = 0.005 * getSimulationPeriod()

    # Called when actor is added to GameGrid
    def reset(self):
        self.px = self.getX()
        self.py = self.getY()
        dx = math.cos(math.radians(self.getDirectionStart()))
        self.vx = self.speed * dx
        dy = math.sin(math.radians(self.getDirectionStart()))
        self.vy = self.speed * dy

    def act(self):
        if isGameOver:
            return
        self.vy = self.vy + g * self.dt
        self.px = self.px + self.vx * self.dt
        self.py = self.py + self.vy * self.dt
        self.setLocation(Location(int(self.px), int(self.py)))
        self.setDirection(math.degrees(math.atan2(self.vy, self.vx)))
        if not self.isInGrid():
            self.removeSelf()
            crossbow.show(0) # Load crossbow

    def collide(self, actor1, actor2):
        actor2.sliceFruit()
        return 0

# ------ End of class definitions --------------------

def keyCallback(e):
    code = e.getKeyCode()
    if code == KeyEvent.VK_UP:
        crossbow.setDirection(crossbow.getDirection() - 5)
    elif code == KeyEvent.VK_DOWN:
        crossbow.setDirection(crossbow.getDirection() + 5)
    elif code == KeyEvent.VK_SPACE:
        if isGameOver:
            return
        if crossbow.getIdVisible() == 1: # Wait until crossbow is loaded
            return
        crossbow.show(1) # crossbow is released
```

```
        dart = Dart(100)
        addActorNoRefresh(dart, crossbow.getLocation(), crossbow.getDirection())
        # for a new dart, the collision partners are all existing fruits
        dart.addCollisionActors(toArrayList(getActors(Fruit)))


FACTORY_CAPACITY = 20
FACTORY_SLOWDOWN = 35
screenWidth = 600
screenHeight = 400
g = 9.81
isGameOver = False

makeGameGrid(screenWidth, screenHeight, 1, False, keyPressed = keyCallback)
setTitle("Use Cursor up/down to target, Space to shoot.")
setBgColor(makeColor("skyblue"))
addStatusBar(30)
factory = FruitFactory.create(FACTORY_CAPACITY, FACTORY_SLOWDOWN)
addActor(factory, Location(0, 0))  # needed to run act()
crossbow = Crossbow()
addActor(crossbow, Location(80, 320))
setSimulationPeriod(30)
doRun()
show()

while not isDisposed() and not isGameOver:
   # Don't show message if same
   oldMsg = ""
   msg = "#hit: "+str(FruitFactory.nbHit)+" #missed: "+str(FruitFactory.nbMissed)
   if  msg != oldMsg:
        setStatusText(msg)
        oldMsg = msg
   if FruitFactory.nbHit + FruitFactory.nbMissed == FACTORY_CAPACITY:
       isGameOver = True
       removeActors(Dart)
       setStatusText("You smashed " + str(FruitFactory.nbHit) + " out of "
       + str(FACTORY_CAPACITY) + " fruits")
       addActor(Actor("sprites/gameover.gif"), Location(300, 200))

   delay(100)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

La plupart des actions utilisateur devraient être interdites après le Game Over. Le plus simple pour implémenter ces restrictions est d'introduire un fanion global *isGameOver = True* utilisé pour effectuer un return conditionnel prématuré dans le gestionnaire des événements clavier et dans la méthode *act()* de la flèche.

L'utilisateur devrait par contre toujours être en mesure de bouger l'arbalète après le Game Over mais pas de tirer des flèches.

EXERCISES

1. Compter le nombre de flèches utilisées durant la partie et fixer un maximum de flèches disponibles. Lorsque toutes les flèches ont été épuisées, le jeu s'arrête également. Ajouter l'information appropriée dans la barre d'état.

2. Ajouter une récompense pour chaque fruit fendu :
   Melon: 5 points
   Orange: 10 points
   Fraise: 15 points

3. Faire en sorte que le jeu redémarre lors d'une pression sur la touche espace après le Game Over. S'assurer que l'utilisateur ne puisse pas redémarrer le jeu involontairement en demandant une confirmation.

4. Étendre ou modifier le jeu selon votre imagination.

Méthodes principales de la classe JGameGrid

Importation du module: from gamegrid import *

Bibliothèque de sprites

Classe GameGrid (fonctions globales lorsque la grille est construite avec *makeGameGrid()*)

Méthode	Action
GameGrid(nbHorzCells, nbVertCells, cellSize, color)	Génère une fenêtre de jeu possédant le nombre indiqué de cellules en horizontal et en vertical. Elles ont une taille de *cellSize*. Les lignes de la grille sont visibles et de couleur *color*
GameGrid(nbHorzCells, nbVertCells, cellSize, color, bgImagePath)	Idem, en utilisant l'image de fond *bgImagePath*
GameGrid(nbHorzCells, nbVertCells, cellSize, None, bgImagePath, False)	Idem, sans afficher les lignes de la grille (*color=None*) ni la barre de navigation (*False*)
act()	Méthode appelée par le système à chaque cycle de simulation après les Actor.act()
addActor(actor, location)	Ajoute l'acteur *actor* à la fenêtre de jeu à la position indiquée
addKeyListener(listener)	Enregistre la fonction de rappel *listener* pour gérer les événements clavier
addMouseListener(listener, mouseEventMask)	Enregistre la fonction de rappel *listener* pour gérer les événements souris
addStatusbar(height)	Ajoute une barre d'état à la fenêtre de jeu
delay(time)	Met le programme en pause pendant *time* millisecondes
doPause()	Met les cycles de simulation en pause
doStep()	Effectue la simulation pas à pas
doReset()	Remet tous les acteurs à leur position initiale et redémarre la simulation
doRun()	Démarre les cycles de simulation
getActors(Actor class)	Retourne sous forme de liste tous les acteurs du jeu appartenant à la classe spécifiée
getBg()	Retourne la référence à *GGBackground*
getBgColor()	Retourne la couleur d'arrière-fond
getKeyCode()	Retourne le code clavier de la dernière touche actionnée
getOneActorAt(location)	Retourne le premier acteur de la cellule indiquée par *location* (zéro s'il n'y a pas d'acteur)
getOneActor(Actor class)	Retourne le premier acteur de la classe *class* (zéro s'il n'y a pas d'acteur de la classe *class*)
getRandomEmptyLocation()	Retourne la position d'une cellule vide choisie au hasard dans la grille de jeu
getRandomLocation()	Retourne la position d'une cellule vide ou occupée choisie au hasard dans la grille de jeu
hide()	Cache la fenêtre de jeu sans la fermer
isAtBorder(location)	Retourne *True* si la cellule *location* est située au bord de la grille de jeu
isEmpty(location)	Retourne *True* si la cellule *location* est vide

isInGrid(location)	Retourne *True* si la cellule est située à l'intérieur de la grille de jeu
kbhit()	Retourne *True* si une touche a été enfoncée
toLocation(x, y)	Retourne la cellule contenant le point de coordonnées *x* et *y* (pixels)
openSoundPlayer("wav/ping.wav")	Prépare la restitution d'un fichier audio. Les sons suivants sont disponibles dans *tigerjython.jar*: bird.wav, boing.wav, cat.wav, click.wav, explore.wav, frog.wav, notify.wav
play()	Joue le son précédemment chargé avec *openSoundPlayer*
refresh()	Rafraîchit la fenêtre de jeu
registerAct(onAct)	Enregistre la fonction de rappel *onAct* qui est appelée à chaque cycle de simulation
registerNavigation(started = onStart, stepped = onStep, paused = onPause, resetted = onReset, periodChanged = onPeriodChange)	Enregistre les fonctions de rappel *onStart*, *onStep*, *onPause*, *onReset*, *onPeriodChange* qui sont appelées lorsque la barre de navigation est visible. Il n'est pas nécessaire de toutes les spécifier.
removeActor (actor)	Supprime l'acteur *actor* de la fenêtre de jeu
removeActorsAt(location)	Supprime tous les acteurs situés dans la cellule *location*
removeAllActors()	Supprime tous les acteurs présents dans la fenêtre de jeu
reset()	Réinitialise la simulation en replaçant les acteurs encore présents sur le jeu à position qu'ils occupaient avant le début de la partie
show()	Afficher la fenêtre de jeu
setBgColor(color)	Règle la couleur d'arrière-plan de la fenêtre de jeu
setSimulationPeriod (milisec)	Règle la période de la boucle de simulation
setStatusTest(text)	Modifie le texte *text* de la barre d'état
setTitle(text)	Ajuste le titre affiché dans la barre de titre de la fenêtre

class Actor

Actor(spritepath)	Génère un acteur associé à l'image de sprite chargée à partir du fichier de chemin *spritepath*
Actor(True, spritepath)	Idem. *True* indique que l'acteur peut subir des rotations
Actor(spritepath, nbSprites)	Génère un acteur avec *nbSprites* images de sprite différentes. Les noms de fichiers images seront composés à partir des *spritepath* en ajoutant un nombre entre *0* et *nbSprites-1* Exemple : index_0, index_1, ou fish_0.gif , fish_1.gif, …
Actor(True, spritepath, nbSprites)	Idem, mais en permettant les rotations
act()	Méthode appelée périodiquement sur chaque acteur après le début du cycle de simulation
addActorCollisionListener(listener)	Enregistre le gestionnaire d'événements de collisions
addCollisionActor(actor)	Enregistre *actor* comme partenaire de collision
addMouseTouchListener (listener)	Enregistre le gestionnaire (fonction de rappel) pour l'événement *MouseTouch*
collide(actor1, actor2)	Fonction de rappel appelée lors d'une collision entre *actor1* et *actor2*. Retourne le nombre de cycles à partir de la collision pendant lesquels les événements de collision entre *actor1*et *actor2* sont ignorés

getCollisionActors()	Retourne une liste de candidats à la collision
getDirection()	Retourne la direction du mouvement de l'acteur
getIdVisible()	Retourne l'ID du sprite actuellement affiché
getNeighbours(distance)	Retourne une liste de tous les acteurs qui sont éloignés de *distance* par rapport à l'acteur sur lequel la méthode est invoquée. Distance définie un cercle autour du centre de la cellule actuelle (unité: largeur de la cellule)
getNextMoveLocation (location)	Prédit les coordonnées de l'acteur après le prochain appel à *move()*
getX()	Retourne la coordonnée horizontale (dans le système de coordonnées grille) de la cellule occupée par l'acteur
getY()	Retourne la coordonnée verticale (dans le système de coordonnées grille) de la cellule occupée par l'acteur
hide()	Cache l'acteur sans le supprimer. Après *reset()* il redevient visible
isInGrid()	Retourne *True* si l'acteur est situé à l'intérieur de la grille de jeu
isHorzMirror()	Retourne *True* si le sprite de l'acteur est retourné horizontalement (inversion gauche-droite)
isVertMirror()	Retourne *True* si le sprite de l'acteur est retourné verticalement (inversion haut-bas)
isMoveValid()	Retourne *True* si l'acteur reste dans la grille de jeu après un appel à *move()*
isNearBorder()	Retourne *True* si l'acteur se situe au bord de la grille de jeu
isVisible()	Returns *True* si l'acteur est visible
move()	Déplace l'acteur dans une cellule adjacente en conservant la direction actuelle
move(distance)	Idem, en spécifiant la distance parcourue par le déplacement
reset()	Méthode appelée lorsque l'acteur est ajouté à la *GameGrid* et lorsque le bouton « reset » est actionné
setCollisionCircle (spriteId,center, radius)	Définit une zone de collision circulaire de centre *center* et de rayon *radius* pour le sprite d'ID *spriteId*
setCollisionLine(spriteId, startPoint, endPoint)	Définit une zone de collision rectiligne définie par les points *startPoint* et *endPoint* pour le sprite d'ID *spriteId*
setCollisionRectangle(spriteId, center, width, height)	Définit une zone de collision rectangulaire de centre *center*, de largeur *width* et de hauteur *height*
setCollisionSpot(spriteId, spot)	Définit une zone de collision en un unique point de coordonnées *spot* pour le sprite d'ID *spriteId*
setCollisionImage(spriteId)	Définit les pixels non-transparents pour la collision. Seulement disponible si le partner utilise spot, line ou circle
setHorzMirror(True)	Effectue une inversion gauche/droite sur le sprite de l'acteur
setVertMirror(True)	Effectue une inversion haut/bas sur le sprite de l'acteur
setSlowDown(factor)	Ralentit d'un facteur *factor* les appels à la méthode *act()* des acteurs
setLocation(location)	Place l'acteur dans la cellule de coordonnées *location* au sein de la grille de jeu

setLocationOffset(point)	Décale le centre de l'image de sprite par rapport au centre de la cellule occupée par l'acteur. Ne change rien à la position de l'acteur dans la grille de jeu
setPixelLocation(location)	Place l'acteur aux coordonnées *location* exprimées en pixels (la position dans la grille et le décalage sont ajustés en conséquence)
setX(x)	Ajuste la coordonnée *x* à la valeur spécifiée
setY(y)	Ajuste la coordonnée *y* à la valeur spécifiée
show()	Rend visible le sprite d'ID 0
show(spriteId)	Rend visible le sprite d'ID *spriteId*
showNextSprite ()	Rend visible la prochaine image de sprite (*spriteId* est incrémenté de 1 modulo *nbSprites*). Ainsi, une fois que l'on a atteint le dernier sprite, on recommence au sprite d'id 0
showPreviousSprite()	Rend visible la précédente image de sprite (*spriteId* est décrémenté de 1 modulo *nbSprites*). Ainsi, une fois que l'on a atteint le premier sprite d'ID 0, en recommence avec le dernier d'ID *nbSprite-1*
removeSelf()	Supprime l'acteur. Il ne réapparaît plus lors d'un prochain *reset()*
reset()	Cette fonction est appelée par *GameGrid.addActor()* et lors d'un clic sur le bouton reset
turn(angle)	Change la direction du mouvement de l'acteur de l'angle indiqué en degrés, dans le sens des aiguilles de la montre

class Location

Location(x, y)	Génère un objet *location* permettant une localisation à partir des coordonnées horizontales et verticales de la cellule au sein de la grille de jeux
Location(location)	Génère un objet *location* à partir de *location* (clone)
clone()	Retourne une nouvelle localisation avec les mêmes coordonnées grille
equals(location)	Retourne *True* si la localisation actuelle de l'acteur correspond à *location*
get4CompassDirectionTo(location)	Retourne une liste comportant les positions des quatre cellules adjacentes Ouest, Est, Nord, Sud par rapport à la position courante
getCompassDirectionTo(location)	Retourne une liste comportant les positions des huit cellules adjacentes, également en diagonale. Donc idem, mais en rajoutant encore Nord-Ouest, Nord-Est, Sud-Est et Sud-Ouest
getDirectionTo(location)	Retourne en degrés la direction à prendre pour aller de la position actuelle vers la position *location* donnée. 0 degrés = Est
getNeighbourLocation(direction)	Retourne la position d'une des huit cellules voisines. C'est la cellule la plus proche de la direction indiquée qui est retournée
getNeighbourLocations(distance)	Retourne une liste de toutes les cellules dont le centre se trouve à une distance inférieure ou égale à *distance* de la cellule actuelle
getX()	Retourne la position horizontale de la cellule occupée par l'acteur sur lequel *getX()* est invoquée
getY()	Retourne la position verticale de la cellule occupée par l'acteur sur lequel *getY()* est invoquée

class GGBackground

clear()	Efface l'arrière-fond actuel en remplaçant tous ses pixels par la couleur d'arrière-plan actuellement en usage
clear(color)	Idem, mais en remplaçant par la couleur d'arrière-plan *color*
drawCircle(center, radius)	Dessine un cercle de centre *center* et de rayon *radius*. Les coordonnées du centre ainsi que le rayon sont indiquées en pixels
drawLine(x1,y1, x2, y2)	Dessine un segment droit entre les points définis par les coordonnées pixel (x1, y1) et (x2, y2).
drawLine(pt1, pt2)	Idem, mais en donnant les coordonnées des points extrémaux sous forme d'objets *Point* (pixels)
drawPoint(pt)	Dessine un point de coordonnées indiquées par l'objet *Point pt* (pixels)
drawRectangle(pt1, pt2)	Dessine un rectangle de sommets supérieur gauche *pt1* et inférieur droit *pt2* (objets *Point* en pixels)
drawText(text, pt)	Écrit le texte *text* à la position indiquée par *pt* (objet *Point* en pixels)
fillCell(location, color)	Remplit la cellule de position *location* (coordonnées grille) avec la couleur *color*
fillCircle(center,radius)	Dessine un cercle plein de centre *center* et de rayon *radius*. Les coordonnées et le rayon sont donnés en pixels
getBgColor()	Retourne la couleur de fond actuelle
getColor(location)	Retourne la couleur de fond présente au centre de la cellule *location*. Les pixels de l'acteur ne sont pas pris en compte
save()	Sauvegarde la couleur de fond actuelle qui peut ensuite être restituée avec *restore()*
setBgColor(color)	Change la couleur d'arrière-fond
setFont(font)	Règle la police de caractères utilisée pour afficher les textes
setLineWidth(width)	Règle la largeur des lignes
setPaintColor(color)	Règle la couleur de dessin à *color*
setPaintMode()	Dessine sans tenir compte de la couleur de l'arrière-fond
setXORMode(color)	Passage en mode de dessin XOR. Deux appels successifs avec la même couleur *color* s'annulent
restore()	Restore l'arrière-fond précédemment sauvegardé avec *save()*.

Documentation en ligne complète de la classe *JGameGrid* : **JGameGridDoc**

chapitre huit

EXPÉRIENCES INFORMATIQUES

Objectifs d'apprentissage

- Être capable de résoudre des problèmes stochastiques simples à l'aide de simulation sur ordinateur impliquant des nombres aléatoires.
- Comprendre que les expériences aléatoires sont sujettes à des fluctuations statistiques. Être en mesure de représenter des résultats sous forme de diagrammes de fréquence et de les interpréter.
- Être capable de tester la signification d'un échantillon à l'aide d'une simulation par ordinateur à l'aide d'un test du chi-carré.
- Être capable d'utiliser l'ordinateur pour simuler des populations.
- Savoir ce qu'est l'ensemble de Mandelbrot et être capable de le représenter graphiquement.
- Connaître la notion de fondamentale et d'harmonique ainsi que le concept de spectre de fréquences.

"Je n'ai foi que dans les statistiques que j'ai moi-même concoctée."

Attribué à Winston Churchill

8.1 SIMULATIONS

INTRODUCTION

Les simulations par ordinateur ne jouent pas seulement un rôle important en recherche et dans l'industrie mais également dans le monde de la finance. Elles sont utilisées pour simuler le comportement d'un système réel complexe à l'aide d'un ordinateur. Comparées aux expériences et études réelles, les simulations informatiques présentent l'avantage d'être bon marché, écologiques et de ne présenter aucun danger. Elles ne peuvent cependant pas refléter parfaitement la réalité pour plusieurs raisons:

- La réalité ne peut jamais être parfaitement représentée par des nombres en raison des erreurs de mesure, sauf pour les problèmes de dénombrement.
- Le plus souvent, les interactions entre les différents composants du système ne sont pas connues parfaitement. Ceci vient du fait que les lois sous-jacentes aux phénomènes ne sont pas exactes [**plus...**] or not all influences are taken into account [**plus...**]

Il n'empêche que les simulations informatiques deviennent de plus en plus précises en raison de l'accroissement de la puissance de traitement des ordinateurs. Il suffit de penser aux prévisions météorologiques pour les prochains jours. Le hasard joue un rôle très important dans nos vies puisque nombre de nos décisions ne sont pas prises sur la base d'arguments purement logiques, mais suite à une estimation intuitive de valeurs probabilistes. Le recours au hasard peut également simplifier drastiquement des problèmes possédant une solution exacte. Il est par exemple extrêmement difficile et pratiquement impossible par un algorithme, en un temps raisonnable, le chemin le plus court entre deux villes A et B sur un réseau routier pourvu de nombreuses connexions [**plus...**]

CONCEPTS DE PROGRAMMATION: *simulation informatique, expérience informatique, fluctuations statistiques*

L'ORDINATEUR COMME PARTENAIRE DE JEU

Votre ami Nora suggère le jeu suivant: « tu peux jeter trois dés et tu gagnes un jeton si tu obtiens un 6. En revanche, si tu n'obtiens aucun 6, je gagne et tu dois me donner un jeton ».

A première vue, vous pensez que le jeu est équitable parce que vous y réfléchissez rapidement et que vous réalisez que, pour chaque dé, la probabilité d'obtenir un 6 vaut 1/6. Vous vous dites alors que la probabilité d'obtenir un 6 au premier, au second, ou au troisième jet vaut 1/6 + 1/6 + 1/6 = 1/2.

Il est tout-à-fait possible de tester la validité de ce raisonnement en utilisant l'ordinateur et vos compétences de programmation. On suppose qu'il revient au même de lancer les trois dés en même temps ou l'un après l'autre. En d'autres termes, on suppose que les jets sont indépendants et que toutes les faces du dé surviennent de manière équiprobables avec probabilité 1/6. Il y a essentiellement deux manières d'attaquer ce problème : **l'approche statistique** et **l'approche combinatoire**. L'approche statistique correspond au jeu réel. On simule les jets de dés en générant **de manière répétée** trois nombres aléatoires compris entre 1et 6 et en dénombrant

les cas gagnants.

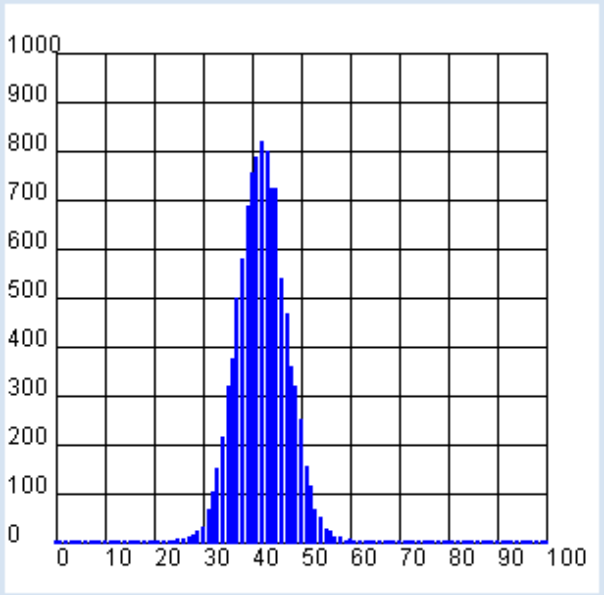
```
from random import randint

n = 1000 # number of games
won = 0
repeat n:
    a = randint(1, 6)
    b = randint(1, 6)
    c = randint(1, 6)
    if a == 6 or b == 6 or c == 6:
        won += 1

print "Won:", won, " of ", n, "games"
print "My winning percentage:", won / n
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

Le résultat de la simulation indique que la probabilité d'un jeu gagnant vaut environ 0.42 et non 0.5 comme on s'y attendait. Cette valeur change bien entendu d'une simulation à l'autre car elle est sujette aux **fluctuations statistiques**. Comme on peut s'y attendre intuitivement, **plus le nombre d'expériences est élevé**, plus le **résultat est précis**.

Les fluctuations statistiques jouent un rôle très important dans les simulations informatiques.

Pour bien comprendre ce phénomène, on reproduit un très grand nombre de fois (disons 10'000 fois) l'expérience consistant à simuler le jeu 100 fois de suite puis on représente dans un **diagramme de fréquences** le nombre de jeux gagnants. Il en résulte une **distribution** en forme de cloche, typique en statistiques.



On utilise un *GPanel* comme fenêtre graphique dans ce programme. On peut également afficher un système de coordonnées avec **drawGrid()**. On implémente une simulation de 100 jeux consécutifs dans la fonction **sim()** qui retourne le nombre de jeux gagnants dont on veut investiguer les fluctuations.

```
from gpanel import *
from random import randint

z = 10000
n = 100

def sim():
    won = 0
    repeat n:
        a = randint(1, 6)
        b = randint(1, 6)
        c = randint(1, 6)
        if a == 6 or b == 6 or c == 6:
            won += 1
    return won

makeGPanel(-10, 110, -100, 1100)
drawGrid(0, 100, 0, 1000)
h = [0] * (n + 1)
title("Simulation started. Please wait...")
repeat z:
```

```
    x = sim()
    h[x] += 1
title("Simulation ended")

lineWidth(2)
setColor("blue")
for x in range(n + 1):
    line(x, 0, x, h[x])
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Le maximum de la distribution se trouve environ à 42 puisque la probabilité de gagner vaut environ 0.42 et que l'on répète le jeu 100 fois. Si vous jouez 100 fois avec Nora, il est possible que vous gagniez plus de 50 fois, bien que la probabilité de gagner ne soit que de 0.42. La probabilité que cela arrive et cependant relativement faible (environ 5%) et le jeu est de ce fait inéquitable. Il faut toujours garder à l'esprit que les expériences par ordinateur impliquant des nombres aléatoires subissent des fluctuations statistiques qui s'atténuent lorsque l'on augmente le nombre d'essais.

Quant à l'approche combinatoire, on laisse l'ordinateur tester toutes les possibilités de jeter trois dés l'un après l'autre. Le premier, le second et le troisième jet peuvent chacun résulter en un nombre compris entre un et six. On forme tous les triplets de nombres possibles dans la boucle imbriquée et l'on compte le nombre de possibilités totales dans la variable *possible* alors que l'on dénombre le nombre de cas gagnants dans la variable *favorable*.

```
possible = 0
favorable = 0
for i in range(1, 7):
    for j in range(1, 7):
        for k in range(1, 7):
            possible += 1
            if i == 6 or j == 6 or k == 6:
                favorable += 1
print "favorable:", favorable, "possible:", possible
print "My winning percentage:", guenstige / possible
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

On voit que le nombre de cas favorables vaut 91 alors que le nombre de cas possibles vaut 216. Cela correspond à une probabilité de jeu gagnant de w = *favorable / possible* = 91/216 = 0.42, Valeur que nous avait déjà livré la simulation informatique.

MATÉRIEL SUPPLÉMENTAIRE

On pourrait bien entendu résoudre ce problème sans utiliser l'ordinateur. Pour cela, on considère les trois événements gagnants E1, E2, E3:

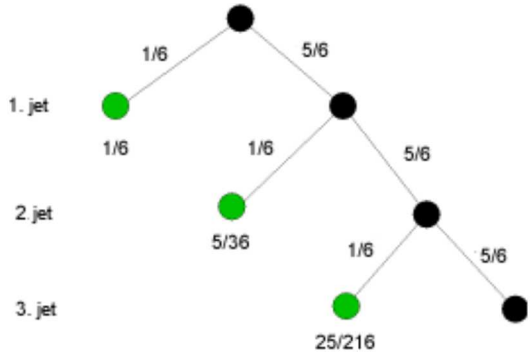
E1: Obtenir un 6 lors du premier jet. Probabilité: 1/6

E2: Pas de 6 lors du premier jet mais lors du deuxième. Probabilité: 5/6 * 1/6

E3: Pas de six au premier ou second jet, mais 6 lors du troisième.
Probabilité: 5/6 * 5/6 * 1/6

Puisque E1, E2, et E3 sont indépendants les uns des autres, la probabilité d'un jeu gagnant vaut la somme des probabilités, à savoir 1/6 + 5/36 + 25/216 = 91/216 = 0.421296.

On peut se représenter le processus dans un arbre:



Il existe encore une solution plus élégante qui consiste à calculer la probabilité de ne pas gagner, à savoir de n'obtenir aucun six du tout. Celle-ci vaut p = 5/6 * 5/6 * 5/6 = 125/216. De ce fait, la probabilité d'un jeu gagnant vaut w = 1 - p = 91/216.

EXERCICES

1. Le Duc Ferdinand de Médicis, de Florence, détermine en l'an 1600 que lors d'un jet de trois dés, il y a le même nombre de possibilités d'obtenir un total de 9 ou de 10 points:

Total de 9 points	**Total de 10 points**
1 + 2 + 6	1 + 3 + 6
1 + 3 + 5	1 + 4 + 5
2 + 2 + 5	2 + 2 + 6
2 + 3 + 4	2 + 3 + 5
3 + 3 + 3	2 + 4 + 4

   Le Duc s'est cependant aperçu qu'il n'y a pas la même probabilité d'obtenir une somme de 9 qu'une somme de 10. Il s'est donc tourné vers Galilée pour trouver une explication.

   Déterminer expérimentalement ces probabilités à l'aide d'une simulation informatique. Utiliser d'abord l'approche statistique puis l'approche combinatoire.

2. Utiliser une simulation informatique pour déterminer la probabilité qu'au moins deux enfants d'une classe de 20 aient leur anniversaire le même jour. On exclut les années bissextiles.

3. À Paris, en 1650, le Chevalier de Méré demanda au mathématicien Blaise Pascal de lui indiquer la probabilité d'occurrence des événements suivants:
   a. Obtenir un 6 parmi 4 jets successifs
   b. Obtenir au moins un double 6 après 24 jets.
   Il croyait les deux événements équiprobables puisque, bien que b) ait une probabilité 6 fois moindre, il y a 6 fois plus d'essais à disposition. Avait-il raison?

4*. Dans le jeu avec Nora, déterminer la probabilité de gagner plus de 50 fois en jouant 100 fois d'affilée.

8.2 POPULATIONS

INTRODUCTION

On utilise fréquemment des simulations informatiques pour prédire le comportement futur d'un système sur la base d'observations ponctuelles ou s'étendant sur un certain laps de temps dans un passé récent. De telles prédictions peuvent être d'une grande importance stratégique et nous avertir suffisamment tôt d'un scénario pouvant conduire à une catastrophe, ce qui permet par exemple d'envisager des mesures préventives. Dans ce domaine, les sujets qui préoccupent beaucoup notre société sont par exemple le changement climatique global et l'augmentation de la population mondiale.

On conçoit une population comme un système d'individus dont le nombre change au fil du temps en fonction de mécanismes internes, d'interactions, et d'influences externes. Si l'on néglige les influences extérieures, on parle de système fermé. Pour bon nombre de populations, la variation de la population au temps *t* est proportionnelle à sa taille au temps *t*. La variation de la valeur actuelle est calculée à partir du taux de variation de la manière suivante :

*nouvelle valeur - ancienne valeur = ancienne valeur * taux de variation * intervalle de temps*

Puisque le membre de gauche représente la différence entre la nouvelle valeur et l'ancienne, cette relation est appelée **équation aux différences**. Le taux d'accroissement peut également être interprété comme une probabilité de croissance par individu et par unité de temps. Si celui–ci est négatif, c'est que la population est en déclin. Le taux d'accroissement peut bien entendu changer au cours du temps.

CONCEPTS DE PROGRAMMATION: *Équation aux différences, taux d'accroissement, croissance exponentielle / limitée, table de mortalité, pyramide des âges, système proie-prédateur*

CROISSANCE EXPONENTIELLE

Les projections concernant les variations de population sont d'un intérêt primordial et peuvent affecter de manière très significative la prise de décisions politiques. Le dernier débat en date est celui de la régulation de la proportion d'étrangers dans la population.

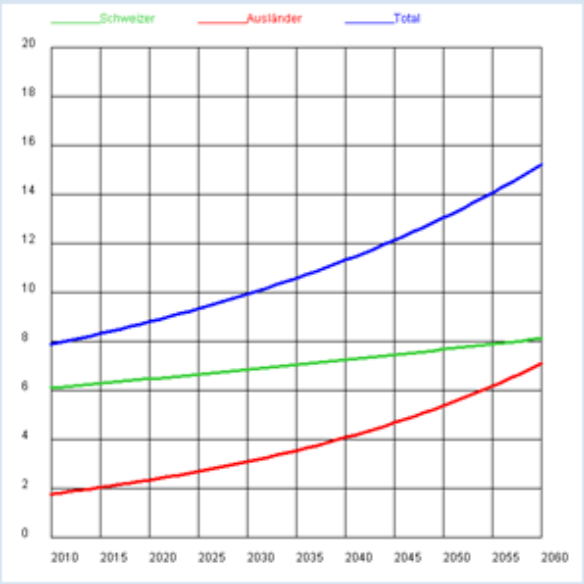
L'OFS (Office Fédéral de la Statistique) publie chaque année le nombre d'habitants en Suisse. Les valeurs pour les années 2010 et 2011 sont les suivantes (source:: **http://www.bfs.admin.ch**, mot-clé: STAT-TAB):

2010: Total z_0 = 7 870 134, parmi lesquels s_0 = 6 103 857 sont suisses
2011: Total z_1 = 7 954 662, parmi lesquels s_1 = 6 138 668 sont suisses

Peut-on échafauder une prédiction de la proportion entre suisses et étrangers pour les 50 prochaines années à partir de cette information ? Il faudrait d'abord calculer le nombre d'étrangers avec $a_0 = z_0 - s_0$ et $a_1 = z_1 - s_0$et, à partir de ces valeurs, trouver le taux d'accroissement annuel entre 2010 et 2011 pour les habitants suisses et les étrangers.

$$rs = \frac{s_1 - s_0}{s_0} = 0.57\% \quad \text{respectivement} \quad ra = \frac{a_1 - a_0}{a_0} = 2.81\%$$

Il n'est maintenant plus très difficile d'estimer la composition de la population pour les 50 prochaines années **à supposer que le taux d'accroissement demeure constant**. On peut le faire à l'aide d'une calculatrice, d'un tableur ou de *Python*. On peut ensuite visualiser les valeurs calculées dans un graphe.




```
from gpanel import *

# source: Swiss Federal Statistical Office, STAT-TAB
z2010 = 7870134 # Total 2010
z2011 = 7954662 # Total 2011
s2010 = 6103857 # Swiss 2010
s2011 = 6138668 # Swiss 2011

def drawGrid():
    # Horizontal
    for i in range(11):
        y = 2000000 * i
        line(0, y, 50, y)
        text(-3, y, str(2 * i))
    # Vertical
    for k in range(11):
        x = 5 * k
        line(x, 0, x, 20000000)
        text(x, -1000000, str(int(x + 2010)))

def drawLegend():
    setColor("lime green")
    y = 21000000
    move(0, y)
    draw(5, y)
    text("Swiss")
    setColor("red")
    move(15, y)
    draw(20, y)
    text("foreigner")
    setColor("blue")
    move(30, y)
    draw(35, y)
    text("Total")

makeGPanel(-5, 55, -2000000, 22000000)
title("Population growth extended")
drawGrid()
drawLegend()

a2010 = z2010 - s2010 # foreigners 2010
a2011 = z2011 - s2011 # foreigners 2011

lineWidth(3)
setColor("blue")
line(0, z2010, 1, z2011)
setColor("lime green")
```

```
line(0, s2010, 1, s2011)
setColor("red")
line(0, a2010, 1, a2011)

rs = (s2011 - s2010) / s2010  # Swiss growth rate
ra = (a2011 - a2010) / a2010  # foreigners growth rate

# iteration
s = s2011
a = a2011
z = s + a
sOld = s
aOld = a
zOld = z
for i in range(0, 49):
    s = s + rs * s  # model assumptions
    a = a + ra * a  # model assumptions
    z = s + a
    setColor("blue")
    line(i + 1, zOld, i + 2, z)
    setColor("lime green")
    line(i + 1, sOld, i + 2, s)
    setColor("red")
    line(i + 1, aOld, i + 2, a)
    zOld = z
    sOld = s
    aOld = a
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Comme le montrent les chiffres, la proportion d'étrangers double entre 2010 et 2035, de sorte qu'elle double en seulement 25 ans et qu'elle quadruple si l'on ajoute encore 25 ans supplémentaires. Il est évident que la taille de la population augmente alors bien plus vite que proportionnellement au temps puisque son taux d'accroissement est constant. Si *T* est le temps nécessaire pour que la population double, la taille *y* après un temps *t* vaut, pour une population de taille initiale *A,*

$$y = A * 2^{1/T}$$

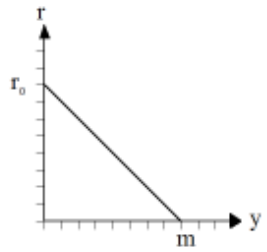
Du fait que la variable temporelle se trouve à l'exposant, cette courbe présente une **croissance exponentielle** extrêmement rapide.

CROISSANCE LIMITÉE

De nombreuses populations habitent un environnement disposant de ressources limitées. La croissance exponentielle rapide due à un taux d'accroissement constant *r* est de ce fait bornée. Cela fait déjà environ 100 ans que le biologiste Carlson a déterminé d'heure en heure les valeurs suivantes (mg) pour une culture de bactéries de levure:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
9.6	18.3	29.0	47.2	71.1	119.1	174.6	257.3	350.7	441.0	513.3	559.7	594.8	629.4	640.8	651.1	655.9	659.8	661.8

On peut comprendre le déroulement de cette expérience à l'aide d'un modèle dans lequel la croissance exponentielle parvient à saturation. On peut considérer que le taux d'accroissement décroit linéairement avec l'augmentation de la population *y* jusqu'à ce qu'il soit nul pour une certaine **valeur de saturation** *m*.



Comme on peut facilement le vérifier avec les substitutions y = 0 et y = m, on obtient la formule suivante:

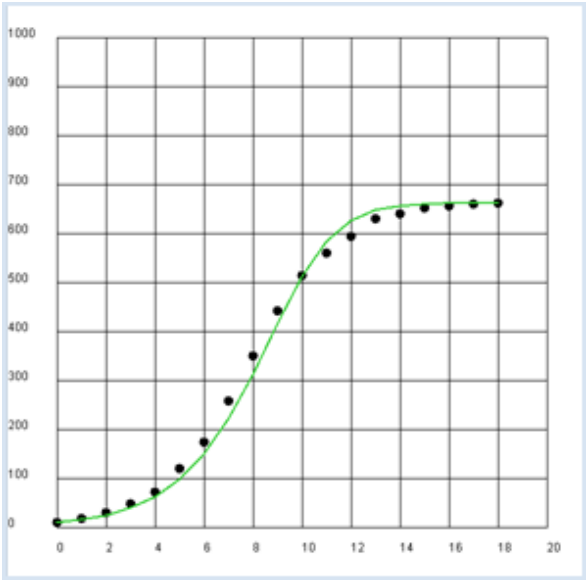
$$r = r_0 * (1 - \frac{y}{m}) = \frac{r_0}{m} * (m - y)$$

En faisant cette hypothèse, on peut représenter graphiquement l'évolution temporelle du processus ainsi que les valeurs expérimentales à l'aide d'un court programme. Pour cela, on itère sur l'équation aux différences que l'on peut écrire

$$dy = y * r * dt = y * r_0 * (1 - \frac{r_0}{m}) * dt$$

avec

dy: nouvelle valeur – ancienne valeur
y: ancienne valeur
dt: intervalle de temps
λ: taux d'accroissement



En utilisant la valeur initiale y0 = 9.6 mg, la valeur de saturation m = 662 mg et le taux d'accroissement initial r0 = 0.62 /h, on obtient une bonne corrélation entre la théorie et l'expérience.

```
from gpanel import *

z = [9.6, 18.3, 29.0, 47.2, 71.1, 119.1, 174.6, 257.3, 350.7, 441.0, 513.3,
559.7, 594.8, 629.4, 640.8, 651.1, 655.9, 659.6, 661.8]

def r(y):
    return r0 * (1 - y / m)

r0 = 0.62
y = 9.6
m = 662

makeGPanel(-2, 22, -100, 1100)
title("Bacterial growth")
drawGrid(0, 20, 0, 1000)
lineWidth(2)
for n in range(0, 19):
    move(n, z[n])
    setColor("black")
    fillCircle(0.2)
    if n > 0:
        dy = y * r(y)
        yNew = y + dy
        setColor("lime green")
```

```
        line(n - 1, y, n, yNew)
        y = yNew
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

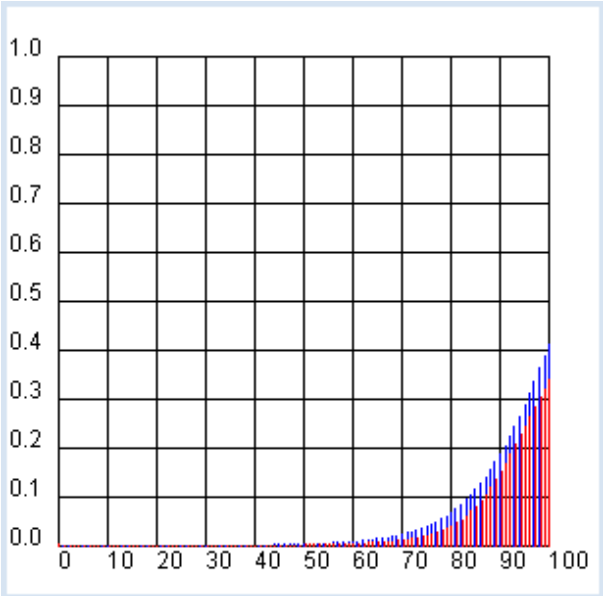
En supposant une diminution linéaire du taux d'accroissement, on obtient une courbe de saturation « en S » typique de l'évolution d'une population. On parle également de **croissance logistique** ou de courbe sigmoïde.

LIFE TABLES

Une manière possible d'estimer la santé d'une population est de considérer la probabilité de dépasser un certain âge ou de mourir à un certain âge. Si l'on veut analyser la distribution des âges au sein de la population suisse, on peut utiliser des données actuelles publiées par l'OFS, à savoir les tables de mortalité (source: **http://www.bfs.admin.ch**, keyword: STAT-TAB).

Ces dernières mettent en évidence les probabilités observées (*qx* et *qy*) pour les hommes et les femmes de mourir à un certain âge, en fonction du sexe. La manière dont ces tables sont construites est relativement simple à comprendre : pour chaque tranche d'âge d'une année (entre 0 et 1 ans, entre 1 et 2 ans, etc.), on considère le nombre de morts survenues l'année dernière chez les hommes d'une part et chez les femmes de l'autre. On divise ensuite chacun de ces nombres par le nombre d'individus total de cette tranche d'âge au début de l'année.

Il est possible de créer un tableau Excel pour la table de mortalité et de copier les colonnes *qx* et *qy* dans un fichier texte *qx.dat* et *qy.dat*. On peut également simplement télécharger ces fichiers depuis **ici** et les copier dans le dossier contenant le programme Python. Le programme peut alors charger ces données et les stocker dans les listes *qx* et *qy*. Puisque ces nombres contiennent parfois des espaces ou des apostrophes pour une meilleure lisibilité, il est nécessaire de les supprimer. On peut alors représenter ces données graphiquement.




```
import exceptions
from gpanel import *

def readData(filename):
    table = []
    fData = open(filename)

    while True:
        line = fData.readline().replace(" ", "").replace("'", "")
        if line == "":
            break
        line = line[:-1] # remove trailing \n
        try:
            q = float(line)
        except exceptions.ValueError:
```

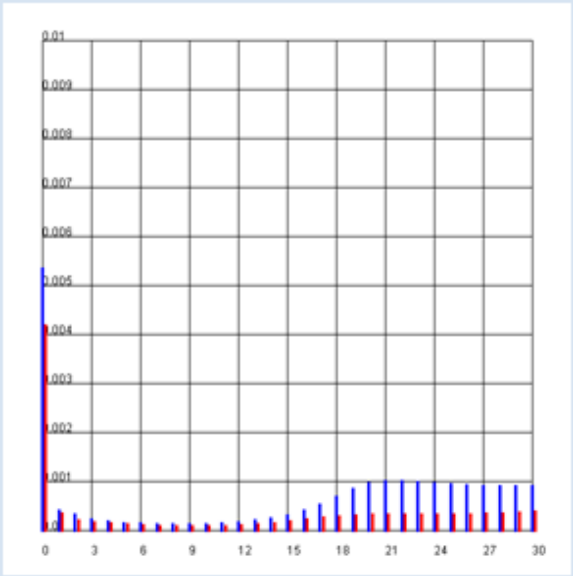
```
            break
        table.append(q)
    fData.close()
    return table

makeGPanel(-10, 110, -0.1, 1.1)
title("Mortality probability (blue -> male, red  -> female)")
drawGrid(0, 100, 0, 1.0)
qx = readData("qx.dat")
qy = readData("qy.dat")
for t in range(101):
    setColor("blue")
    p = qx[t]
    line(t, 0, t, p)
    setColor("red")
    q = qy[t]
    line(t + 0.2, 0, t + 0.2, q)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

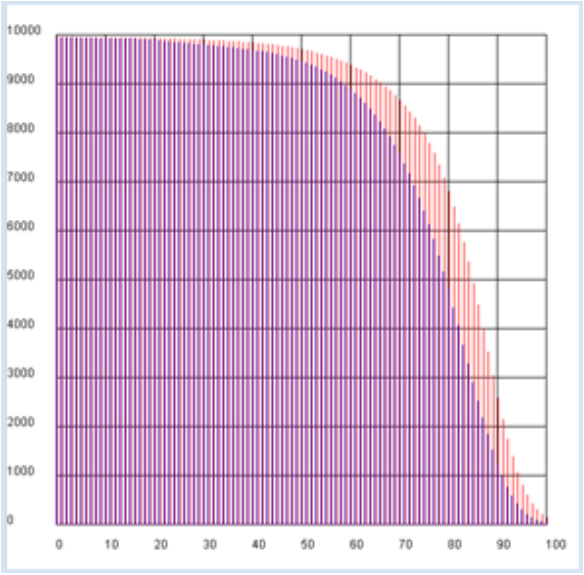
La courbe montre clairement que les femmes vivent plus longtemps que les hommes en moyenne. Le cours des événements durant les 30 premières années de la vie est également intéressant.

Un nombre significativement plus élevé de garçons meurent durant leur première année de vie ainsi qu'entre 15 et 30 ans. Essayez de trouver des explications à ces observations.



ÉVOLUTION TEMPORELLE D'UNE POPULATION

À l'aide des tables de mortalité et d'un programme informatique, on peut traiter de nombreuses questions démographiques intéressantes d'une manière scientifiquement correcte. Dans l'exemple suivant, on examine la manière dont une population de 10 000 nouveau-nés va évoluer sur les 100 prochaines années. On utilise pour ce faire les valeurs *qx* et *qy* comme des taux d'accroissements négatifs.



```
import exceptions
from gpanel import *

n = 10000 # size of the population

def readData(filename):
    table = []
    fData = open(filename)

    while True:
        line = fData.readline().replace(" ", "").replace("'", "")
        if line == "":
            break
        line = line[:-1] # remove trailing \n
        try:
            q = float(line)
        except exceptions.ValueError:
           break
        table.append(q)
    fData.close()
    return table

makeGPanel(-10, 110, -1000, 11000)
title("Population behavior/predictions  (blue -> male, red -> female)")
drawGrid(0, 100, 0, 10000)
qx = readData("qx.dat")
qy = readData("qy.dat")
x = n # males
y = n # females
for t in range(101):
    setColor("blue")
    rx = qx[t]
    x = x - x * rx
    line(t, 0, t, x)
    setColor("red")
    ry = qy[t]
    y = y - y * ry
    line(t + 0.2, 0, t + 0.2, y)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

ESPÉRANCE DE VIE DES FEMMES ET DES HOMMES

Il apparaît clairement de la précédente analyse que les femmes vivent plus longtemps que les hommes. On peut également exprimer cette différence à l'aide d'une seule grandeur appelée **espérance de vie**. Il s'agit de l'âge moyen atteint par les femmes et par les hommes.

Rappelons-nous brièvement la manière dont une moyenne, par exemple la moyenne des notes d'une classe, est définie : on calcule la somme *s* des notes de l'ensemble des étudiants que l'on divise par le nombre *n* d'étudiants. Par souci de simplicité, supposons que toutes les notes sont des nombres entiers compris entre 1 et 6, de sorte que l'on peut calculer *s* de la manière suivante:

s = nombre d'étudiants avec note 1 * 1 + nombre d'étudiants avec note 2 * 2 + ... nombre d'étudiants avec note 6 * 6

ou, de manière plus générale:

moyenne = somme sur (fréquences valeur * valeur) divisé par le nombre total

Si on lit les fréquences à partir d'une distribution de fréquences *h* de la valeur *x* (dans notre cas, il s'agit des notes entre 1 et 6), on utilise plutôt le terme d'**espérance mathématique** pour désigner la moyenne et on peut écrire

$$E = x_1 * h_1 + x_2 * h_2 + \ldots + x_n * h_n$$

$$h_1 + h_2 + ... + h_n$$

Comme vous pouvez le constater, les fréquences *hi* sont **pondérées dans la somme**par la valeur x_i

L'espérance de vie n'est rien d'autre que l'espérance mathématique de l'âge auquel les femmes et les hommes meurent. Pour calculer cette valeur à l'aide d'une simulation informatique, on commence avec un certain nombre d'hommes et de femmes (n = 10000) et on détermine le nombre d'hommes (*hx*)et de femmes (*hy*) qui meurt entre l'âge *t* et *t+1*. Evidemment ces nombres peuvent être exprimés de la manière suivante en utilisant la taille au temps *t* de la population *x* et *y* livrés par le programme précédent ainsi que les taux de mortalité *rx* et *ry* des hommes, respectivement des femmes:

$$hx = x * rx \text{ bzw. } hy = y * ry$$

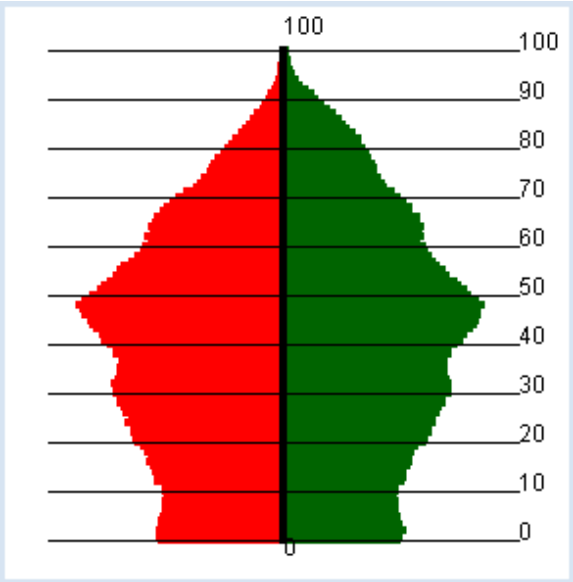
```
n = 10000 # size of the population

def readData(filename):
    table = []
    fData = open(filename)

    while True:
        line = fData.readline().replace(" ", "")
        if line == "":
            break
        line = line[:-1] # remove trailing \n
        try:
            q = float(line)
        except exceptions.ValueError:
           break
        table.append(q)
    fData.close()
    return table

qx = readData("qx.dat")
qy = readData("qy.dat")
x = n
y = n
xSum = 0
ySum = 0
for t in range(101):
    rx = qx[t]
    x = x - x * rx
    mx = x * rx # male deaths
    xSum = xSum + mx * t # male sum
    ry = qy[t]
    y = y - y * ry
    my = y * ry # female deaths
    ySum = ySum + my * t # female sum

print "Male life expectancy:", xSum / 10000
print "Female life expectancy:", ySum / 10000
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

Les données de la population suisse révèlent une espérance de vie de 76 ans pour les hommes et de 81 ans pour les femmes.

PYRAMIDE DES ÂGES

Dans les études démographiques, on regroupe souvent la population par tranches d'âge d'une année à partir desquelles on forme un diagramme de fréquences. Si l'on veut comparer deux groupes de la population, on représente les fréquences de l'un des groupes à gauche et de l'autre à droite. L'usage de cette méthode pour comparer les hommes et les femmes donnent lieu à de magnifiques pyramide des âges.



Le graphique ci-contre a été formé à partir des données du 31 Décembre 2012 que l'on peut trouver sur le site de l'OFS (**http://www.bfs.admin.ch**, keyword: STAT-TAB). Il suffit de copier les données depuis le tableau excel dans les fichiers de test *zx.dat* et *zy.dat*. Il est également possible de les télécharger avec **ici**.

```
import exceptions
from gpanel import *

def readData(filename):
    table = []
    fData = open(filename)
    while True:
        line = fData.readline().replace(" ", "").replace("'", "")
        if line == "":
            break
        line = line[:-1] # remove trailing \n
        try:
            q = float(line)
        except exceptions.ValueError:
           break
        table.append(q)
    fData.close()
    return table

def drawAxis():
    text(0, -3, "0")
    line(0, 0, 0, 100)
    text(0, 103, "100")

makeGPanel(-100000, 100000, -10, 110)
title("Population pyramid (green -> male, red ->  female)")
lineWidth(4)
zx = readData("zx.dat")
zy = readData("zy.dat")
for t in range(101):
    setColor("red")
    x = zx[t]
    line(0, t, -x, t)
    setColor("darkgreen")
    y = zy[t]
    line(0, t, y, t)
setColor("black")
drawAxis()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

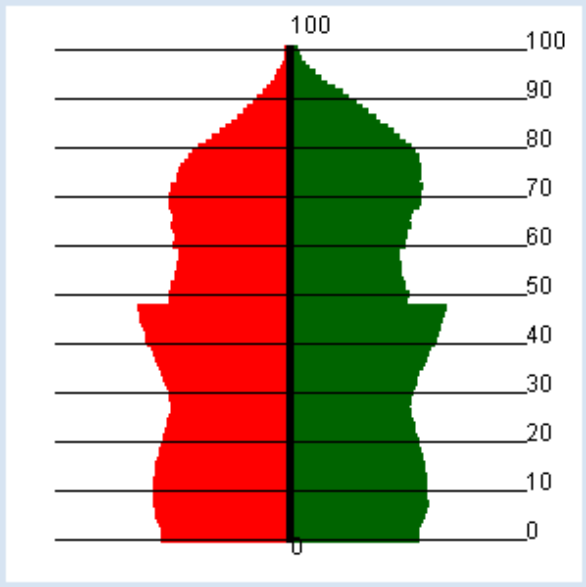
MEMENTO

On repère facilement les baby-boomers nés dans les années 1955 – 1965 (entre 47 et 57 ans).

MODIFICATION DE LA DISTIBUTION DES ÂGES

Une analyse du changement de distribution des âges par décennie peut révéler des informations intéressantes sur les changements vécus par une société. Sur la base de la structure actuelle de la pyramide des âges, il est possible de simuler la distribution des âges sur les cent prochaines années sous les conditions suivantes :

- Il n'y a pas d'immigration ni d'émigration (société fermée)
- On tient compte des morts d'après les tables de mortalité
- Toutes les femmes entre 20 et 39 ans vont enfanter un nombre déterminé d'enfants *k* (les filles et les garçons sont considérés comme équiprobables). On suppose que k = 2.



Le programme permet d'avancer d'une année par une simple pression de n'importe quelle touche du clavier.

```
import exceptions
from gpanel import *

k = 2.0

def readData(filename):
    table = []
    fData = open(filename)
    while True:
        line = fData.readline().replace(" ", "").replace("'", "")
        if line == "":
            break
        line = line[:-1] # remove trailing \n
        try:
            q = float(line)
        except exceptions.ValueError:
           break
        table.append(q)
    fData.close()
    return table

def drawAxis():
    text(0, -3, "0")
    line(0, 0, 0, 100)
    text(0, 103, "100")
    lineWidth(1)
    for y in range(11):
        line(-80000, 10* y, 80000, 10 * y)
        text(str(10 * y))

def drawPyramid():
```

```
    clear()
    title("Number of children: " + str(k) + ", year: " + str(year) +
          ", total population: " + str(getTotal()))
    lineWidth(4)
    for t in range(101):
        setColor("red")
        x = zx[t]
        line(0, t, -x, t)
        setColor("darkgreen")
        y = zy[t]
        line(0, t, y, t)
    setColor("black")
    drawAxis()
    repaint()

def getTotal():
    total = 0
    for t in range(101):
        total += zx[t] + zy[t]
    return int(total)

def updatePop():
    global zx, zy
    zxnew = [0] * 110
    zynew = [0] * 110
    # getting older and dying
    for t in range(101):
        zxnew[t + 1] = zx[t] - zx[t] * qx[t]
        zynew[t + 1] = zy[t] - zy[t] * qy[t]
    # making a baby
    r = k / 20
    nbMother = 0
    for t in range(20, 40):
        nbMother += zy[t]
    zxnew[0] = r / 2 * nbMother
    zynew[0] = zxnew[0]
    zx = zxnew
    zy = zynew

makeGPanel(-100000, 100000, -10, 110)
zx = readData("zx.dat")
zy = readData("zy.dat")
qx = readData("qx.dat")
qy = readData("qy.dat")
year = 2012
enableRepaint(False)
while True:
    drawPyramid()
    getKeyWait()
    year += 1
    updatePop()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

On observe que le futur de la population dépend sensiblement du nombre *k*. Même avec la valeur *k = 2,* la population va décroître sur le long terme.

Pour éviter le clignotement de l'écran lors de la pression sur la touche du clavier, il faut désactiver le rendu automatique à l'aide de l'appel **enableRepaint(False)**. Dans la fonction **drawPyramid()** l'appel *clear()* ne va alors supprimer le graphique que de la mémoire tampon hors écran. Le rendu ne sera alors effectué à l'écran qu'après la fin du calcul de **repaint()**.

EXERCICES

1. Une population est formée de deux individus au temps *t=0*. Chaque année celle–ci augmente avec un taux de naissance de 10% (nombre de naissances par année et par individu). Réaliser un programme Python qui simule l'évolution de cette population sur les 100 premières années et affiche le résultat graphiquement dans un graphique à barres.

2a. Dans une population non vieillissante, la probabilité de mortalité demeure toujours identique, indépendamment de l'âge. Il n'existe pas de telle population d'êtres vivants mais les atomes (noyaux radioactifs) présentent exactement ce comportement. Au lieu d'appeler ceci la probabilité de mortalité, on parle de **probabilité de désintégration**. Réaliser un programme Python qui simule une population de 10'000 noyaux radioactifs dont la probabilité de désintégration vaut 0.1. Étendre la simulation sur une durée de 100 ans et afficher le résultat sous forme de graphique à barres.

2b. Dans le diagramme obtenu, représenter par des lignes verticales l'époque à laquelle la population se réduit à environ 1/2, 1/4, 1/8 et 1/16 de la population initiale. Quelle hypothèse peut-on faire à partir de ce graphique ?

2c* La désintégration radioactive a lieu selon la loi suivante:

$$N = N_0 * e^{-\lambda t}$$

N_0: nombre de radionucléides à l'instant t = 0
N: nombre de radionucléides à l'instant t
λ: carie probabilité par unité de temps (constante de désintégration)

Entrez la meilleure forme possible de la courbe dans le 2a graphique.

3. L'espérance de vie peut également être calculée par une simulation informatique. Pour cela, on simule la vie d'un seul individu d'année en année. Chaque année, l'ordinateur choisit un nombre aléatoire entre 0 et 1 et l'individu meurt cette année-là si le nombre obtenu est inférieur à la probabilité de mortalité *q*. On comptabilise alors le nombre d'années atteint. Une fois cette simulation effectuée pour 10'000 individus, on divise le nombre total d'années de vie obtenu par 10'000. Utiliser cette méthode pour déterminer l'espérance de vie d'une femme en utilisant les valeurs du fichier *qy.dat*.

MATÉRIEL SUPPLÉMENTAIRE

SYSTÈME PROIE-PRÉDATEUR

Il est très intéressant de considérer le comportement de deux populations vivant dans un même écosystème et interagissant l'une avec l'autre. Considérons le scénario suivant : des lapins et des renards cohabitent dans un territoire fermé. Les lapins se multiplient à un taux constant *rx*. Si un renard croise un lapin il y a une certaine probabilité qu'il le mange. À leur tour, les renards présentent un taux de mortalité *ry* et leur taux de croissance est déterminé par leur consommation de lapins.

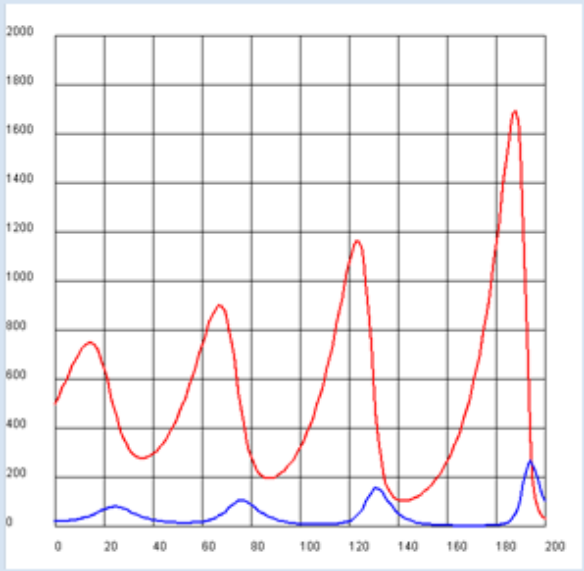
En supposant que la probabilité qu'un renard croise un lapin soit proportionnelle au produit entre le nombre de lapins et le nombre de renards, on obtient deux équations aux différences pour *x* et *y* [**plus...**].

xNew - x = rx * x - gx * x * y
yNew - y = -ry * y + gy * x * y

Utilisons les valeurs:

rx = 0.08
ry = 0.2
gx = 0.002
gy = 0.0004

et une population initiale de *x = 500* lapins et *y = 20* renards. Pour le moment, effectuons la simulation sur une période de 200 générations.




```
from gpanel import *

rx = 0.08
ry = 0.2
gx = 0.002
gy = 0.0004

def dx():
    return rx * x - gx * x * y

def dy():
    return -ry * y + gy * x * y

x = 500
y = 20

makeGPanel(-20, 220, -200, 2200)
title("Predator-Prey system (red: bunnies, blue: foxes)")
drawGrid(0, 200, 0, 2000)
lineWidth(2)
for n in range(200):
    xNew = x + dx()
    yNew = y + dy()
    setColor("red")
    line(n, x, n + 1, xNew)
    setColor("blue")
    line(n, y, n + 1, yNew)
    x = xNew
    y = yNew
```

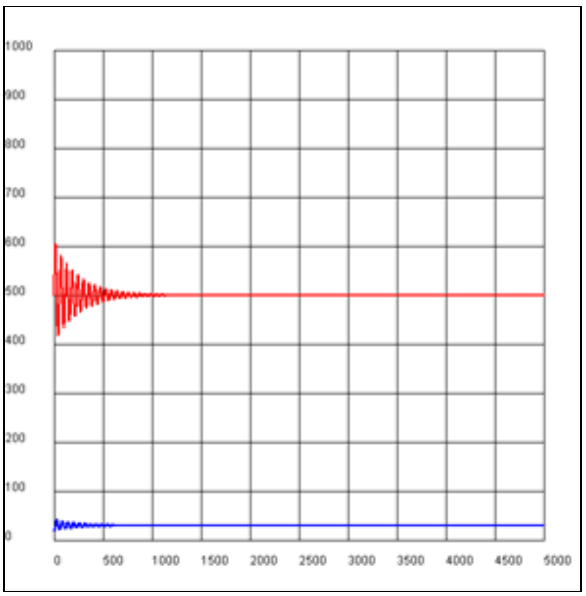
Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

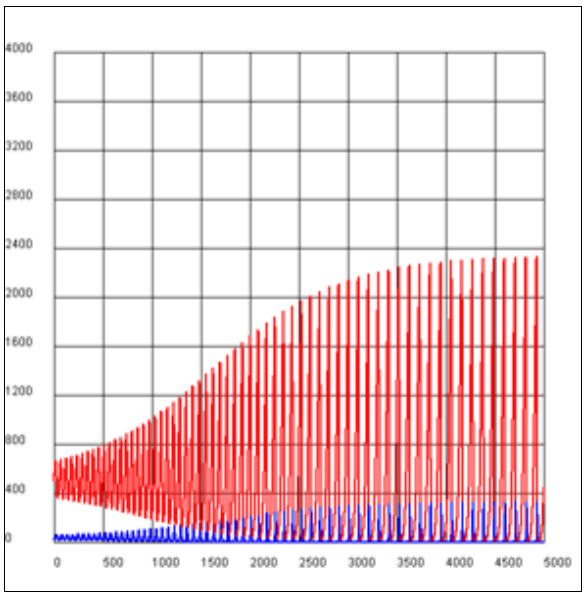
Le nombre de lapins et de renards est sans arrêt en train de fluctuer. Qualitativement, ce processus cyclique peut être interprété de la manière suivante : puisque les renards mangent les lapins, leur population est en nette croissance lorsqu'il y a beaucoup de lapins. Puisque cela a tendance à décimer la population de lapins, la reproduction des renards ralentit. Ce déclin des renards permet aux lapins de se reproduire à nouveau, pratiquement au-delà de toute limite.

EXERCICES

1. Introduire une limite à l'habitat des lapins avec une croissance logistique en prenant un taux d'accroissement de *rx' = rx(1 – x/m)* et en conservant les autres valeurs utilisées dans l'exemple précédent. Montrer que pour m = 2000, les oscillations décroissent avec le temps alors qu'avec m = 3500, ces dernières sont régulières.

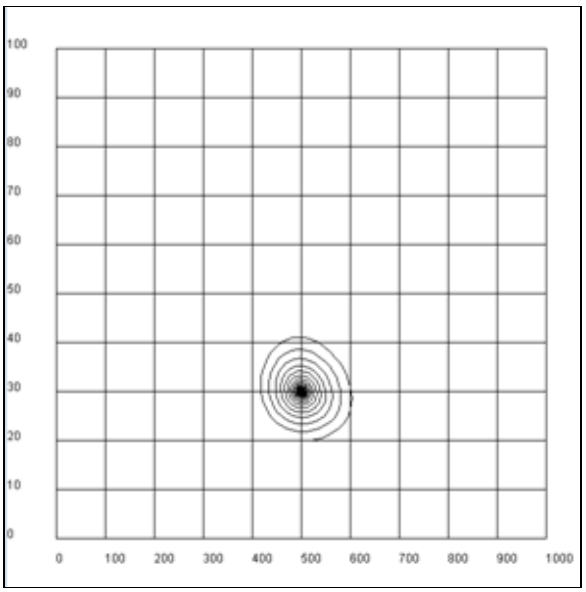


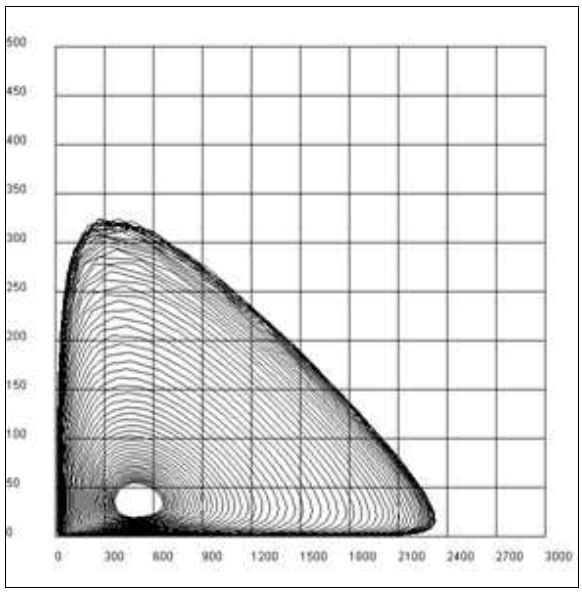
m = 2000, les oscillations s'amenuisent jusqu'à disparaître



m = 3500, l'oscillation est stable

2. Un diagramme dans lequel les tailles des populations sont représentées l'une par rapport à l'autre est appelé un **diagramme de phase**. Développer un programme qui dessine le diagramme de phase pour chacune des situations traitées dans l'exercice précédent. Êtes-vous en mesure de comprendre le comportement qu'ils mettent en évidence ?





8.3 HYPOTHÈSES ET TESTS STATISTIQUES

INTRODUCTION

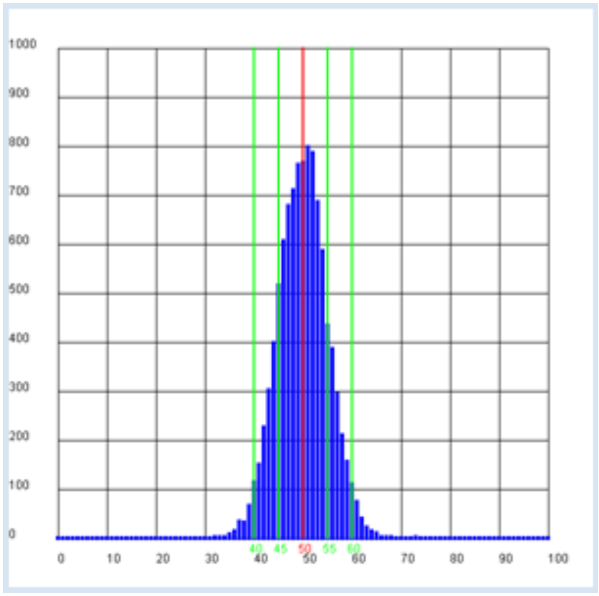
Dans un jeu consistant à tirer à pile ou face avec une pièce de monnaie, vous faites l'hypothèse (on l'appelle **hypothèse nulle**) que cette dernière est truquée, à savoir que les deux faces ne sont pas équiprobables. Vous utilisez un dé pour jouer et vous faites l'hypothèse qu'il n'est pas truqué, ce qui revient à dire que toutes ses faces apparaissent avec la même probabilité ($p = 1/6$). Dans ce chapitre, nous allons étudier une méthode permettant de tester ce genre d'hypothèses. Cela ne peut cependant pas se faire de manière absolument certaine puisque le principe consiste à rejeter l'hypothèse en limitant la probabilité de se tromper en la rejetant à 5% (risque de première espèce).

CONCEPTS DE PROGRAMMATION: *Hypothèse nulle, niveau de confiance, dispersion, test du Chi-carré*

UNE PIÈCE SIGNIFICATIVEMENT BIAISÉE

On pose l'hypothèse nulle que la pièce n'est pas biaisée et, en effectuant $n = 100$ jets, on obtient k fois pile et n-k fois face.

On répète cette expérience de très nombreuses fois, disons $n = 10'000$ fois, ce qui donne lieu à la distribution pour k que l'on peut déterminer avec une simulation. Comme attendu, cela donne lieu à une courbe en cloche autour de la moyenne $m=50$ [**plus...**].



On se pose maintenant la question intéressante de savoir dans quelle plage de valeurs +/- s autour de la moyenne on retrouve un pourcentage donné de tests, par exemple 68%. Dans notre cas, $s = 5$ et environ 68% des expériences ont une valeur entre 45 et 55. On peut également déterminer le paramètre de **dispersion** dans la simulation informatique en ajoutant les fréquences à gauche et à droite de la moyenne jusqu'à atteindre 6800 qui correspond aux 68% de 10'000.

Le programme représente en plus le domaine de valeurs comprenant 95% de tous les résultats et met en évidence que cela correspond environ au double de la dispersion (ici entre 40 et 60).

```
from gpanel import *
import random

n = 100 # size of the test  group
p = 0.5
z = 10000

def showDistribution():
    setColor("blue")
```

```
    lineWidth(4)
    for t in range(n + 1):
        line(t, 0, t, h[t])

def showMean():
    global mean
    sum = 0
    for t in range(n + 1):
        sum += h[t] * t
    mean = int(sum / z + 0.5)
    setColor("red")
    lineWidth(2)
    line(mean, 0, mean, 1000)
    text(mean - 1, -30, str(mean))

def showSpreading(level):
    sum = h[mean]
    for s in range(1, 20):
        sum += h[mean + s] + h[mean - s]
        if sum > z * level:
            break
    setColor("green")
    lineWidth(2)
    line(mean + s, 0, mean + s, 1000)
    text(mean + s - 1, -30, str(mean + s))
    line(mean - s, 0, mean - s, 1000)
    text(mean - s - 1, -30, str(mean - s))

def sim():
    sum = 0
    repeat n:
       w = random.random()
       if w < p:
           sum +=1
    return sum

makeGPanel(-0.1 * n, 1.1 * n, -100, 1100)
title("Coin toss,  distribution of number")
drawGrid(0, n, 0, 1000)
h = [0] * (n + 1)

repeat z:
    k = sim()
    h[k] += 1

showDistribution()
showMean()
showSpreading(0.68)
showSpreading(0.95)
```

Prorammcode markieren (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Si l'on répète un très grand nombre de fois l'expérience consistant à lancer une pièce de monnaie non biaisée 100 fois d'affilée, le nombre de jets « pile » obtenus sera compris dans l'intervalle [50-5, 50+5] dans 68 % des cas et dans l'intervalle [50-10, 50+10] dans 95% des cas [**plus...**] .

Si l'on effectue un test avec une pièce de monnaie et que l'on obtient un nombre de jets « face » supérieur à 60 ou inférieur à 40, on rejette l'hypothèse que la pièce n'est pas biaisée. En d'autres termes, on admet que la pièce est biaisée. Dans ce cas, on pourrait rejeter l'hypothèse de manière erronée avec une probabilité de 5% (l'intervalle de confiance du test). Par souci de concision, on se contente parfois de dire que **la pièce est significativement biaisée**.

UN DÉ SIGNIFICATIVEMENT BIAISÉ

On considère un dé dont on veut tester s'il est non truqué, à savoir si toutes ses faces ont la même probabilité 1/6 de sortir. On pose l'hypothèse nulle que le dé n'est pas baisé.

Il nous faudra dans cet exemple utiliser une méthode légèrement différente de celle vue pour la pièce de monnaie puisque chaque jet possède 6 issues possibles et non seulement deux, à savoir les entiers compris entre 1 et 6. Il est clair que pour obtenir un résultat significatif, il faudra lancer le dé un grand nombre de fois, disons 600 fois, et reporter les fréquences d'apparition des faces dans un tableau:

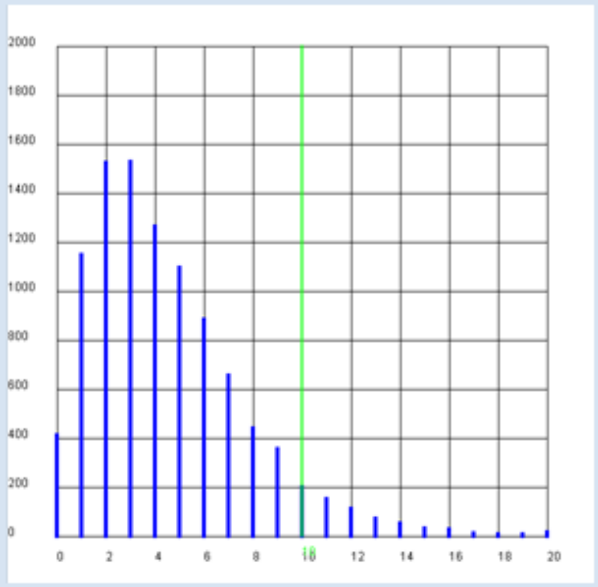
Résultat du dé	Fréquence observée (u)	Fréquence théorique (valeur attendue e)
1	112	100
2	128	100
3	97	100
4	103	100
5	88	100
6	72	100
Total	600	600

Fréquences observées et théoriques

Pour introduire une mesure de l'écart entre les valeurs observées et les valeurs théoriques découlant de l'hypothèse que la pièce n'est pas biaisée, il faut calculer pour chaque face l'écart au carré relatif $(u - e)^2 / e$ et ajouter toutes ces valeurs. Ce résultat est appelé le χ^2 (prononcer "khi-carré ").

Cela soulève la question de savoir à quoi ressemble la distribution de fréquences du χ^2, à savoir le nombre d'occurrences des différentes valeurs du χ^2si l'on répète l'expérience un grand nombre de fois. Pour cela, on effectue une autre simulation informatique comportant 10'000 échantillons dont on détermine la distribution. Pour se simplifier la vie, on arrondit le χ^2 à des valeurs entières [**plus...**].

On fixe à nouveau la valeur critique du χ^2, en-dessous de laquelle se trouvent 95% des valeurs obtenues. D'après la simulation, on a que $s = 11$ [**plus...**].




```
from gpanel import *
import random

n = 600 # number of tosses
p = 1 / 6
z = 10000

def showDistribution():
    setColor("blue")
    lineWidth(4)
    for i in range(21):
        line(i, 0, i, h[i])

def showLimit(level):
```

```
    sum = 0
    for i in range(21):
        sum += h[i]
        if sum > z * level:
            break
    setColor("green")
    lineWidth(2)
    line(i, 0, i, 2000)
    text(i, -80, str(i))
    return i

def chisquare(u):
    chisquare = 0
    e = n * p
    for i in range(1, 7):
        chisquare += ((u[i] - e) * (u[i] - e)) / e
    return chisquare

def sim():
    u = [0] * 7
    repeat n:
        t = random.randint(1, 6)
        u[t] += 1
    return chisquare(u)

makeGPanel(-2, 22, -200, 2200)
title("Chi-square simulation  is being carried out. Please wait...")
drawGrid(0, 20, 0, 2000)
h = [0] * 21

repeat z:
    c = int(sim())
    if c < 20:
        h[c] += 1
    else:
        h[20] += 1

title("Chi-square test on  the die")
showDistribution()
s = showLimit(0.95)

# Observed series
u1 = [0, 112, 128, 97, 103, 88, 72]
u2 = [0, 112, 108, 97, 113, 88, 82]
c1 = chisquare(u1)
c2 = chisquare(u2)
print "Die with", u1, "Xi-square:", c1, "loaded?", c1 > s
print "Die with", u2, "Xi-square:", c2, "loaded?", c2 > s
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

La simulation informatique montre les résultats suivants : dans 95% des cas, χ^2 st inférieur ou égal à la valeur critique 11 pour une pièce non biaisée. De ce fait, on a trouvé une méthode pour tester si un dé est pipé : il suffit de calculer le χ^2des fréquences observées. Si la valeur obtenue est supérieure à 11, on peut affirmer avec une probabilité de se tromper de 5% que l'hypothèse nulle qu'il soit équilibré est fausse, et de ce fait, que le dé est pipé.

Les fréquences issues du tableau ci-dessus donnent $\chi 2 = 18.7$. En d'autres termes, il y a une très forte probabilité que le dé en question soit pipé. Si, en faisant la même expérience avec un autre dé, on obtient les fréquences empiriques u2 = [112, 108, 97, 113, 88, 82] et vu que dans ce cas, $\chi 2 = 8.5$, il y a une faible probabilité que le dé en question soit pipé.

DIFFÉRENCES DE COMPOREMENT CHEZ L'HUMAIN

On peut également appliquer le test du χ^2 l'étude du comportement de deux populations humaines. Une question intéressante qui survient souvent est de savoir si, dans un contexte particulier, le comportement des femmes et des hommes est statistiquement différent ou si les deux sexes se comportent de manière identique.

Imaginons que l'on veuille faire une étude sur l'utilisation de Facebook dans une école secondaire. On demande à 106 filles et 86 garçons de cette école s'ils possèdent un compte Facebook. Le résultat de l'enquête sont les suivants:

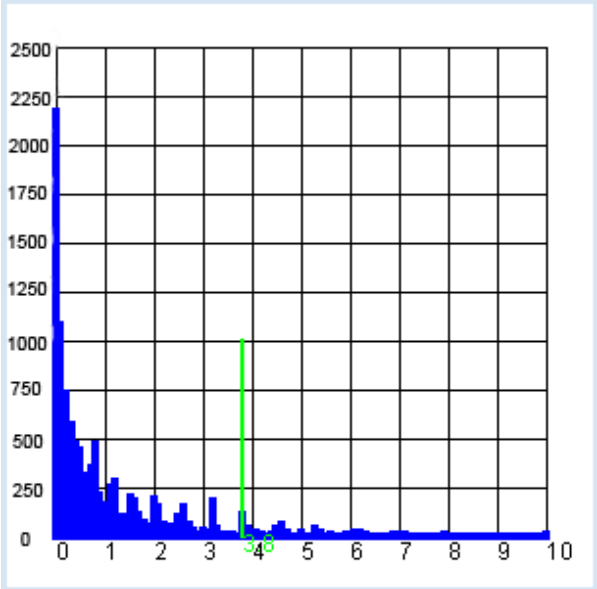
	Facebook Oui	Facebook Non	Total	% Oui
Femmes	87	19	106	82.0%
Hommes	62	24	86	72.1%
Total	149	43	192	77.7%

On remarque que le pourcentage de personnes qui ont un compte Facebook est nettement plus grand chez les femmes que chez les hommes. Mais il faut encore s'assurer que cette probabilité plus élevée chez les femmes soit vraiment statistiquement significative.

Pour effectuer la simulation, en commence par déterminer la probabilité *p* de posséder un compte à partir du nombre total d'hommes et de femmes:

p = (femmes_oui + hommes_oui) / n

On utilise ensuite cette valeur pour simuler le nombre de femmes inscrites sur Facebook en utilisant des nombres aléatoires ainsi que le nombre total de femmes. On obtient alors le nombre *f0* de femmes inscrites et le nombre *f1* de femmes non inscrites. On cherche également les nombres *m0* d'hommes inscrits et *m1* d'hommes non-inscrits. Ces valeurs forment les valeurs *u* servant au calcul du χ^2.



$$\chi^2 = somme\ des\ (u - e)^2 / e$$

Il faut encore déterminer la valeur attendue *e* pour chacun des quatre cas possibles. On suppose que la probabilité totale du "oui" est p = (f0 + m0) / n et, de manière correspondante, que 1-p est la probabilité totale du " Non". On calcule donc:

Valeur attendu pour femmes oui :	ef0	= nombre total de femmes * p
Valeur attendu pour hommes oui :	em0	= nombre total d'hommes * p
Valeur attendu pour femmes non:	ef1	= nombre total de femmes * (1 - p)
Valeur attendu pour hommes non:	em1	= nombre total d'hommes * (1 - p)

Le reste du programme demeure pratiquement inchangé par rapport au test avec le dé.

```
from gpanel import *
import random

z = 10000
# survey values/polls
females_yes = 87
females_no = 19
males_yes = 62
males_no = 24
```

```
def showDistribution():
    setColor("blue")
    lineWidth(4)
    for i in range(101):
        line(i/10, 0, i/10, h[i])

def showLimit(level):
    sum = 0
    for i in range(101):
        sum += h[i]
        if sum > level * z:
            break
    setColor("green")
    lineWidth(2)
    limit = i / 10
    line(limit, 0, limit, 1000)
    text(limit, -80, str(limit))
    return limit

def chisquare(f0, f1, m0, m1):
    # f: females, m: males, 0:yes, 1:no
    w = (f0 + m0) / n # probability of a yes
    # expected value
    ef0 = (f0 + f1) * w # females-yes
    em0 = (m0 + m1) * w # males-yes
    ef1 = (f0 + f1) * (1 - w) # females-no
    em1 = (m0 + m1) * (1 - w) # males-no
    # add up deviations (u - e)*(u - e) / e
    chisquare = (f0 - ef0) * (f0 - ef0) / ef0 \
              + (m0 - em0) * (m0 - em0) / em0 \
              + (f1 - ef1) * (f1 - ef1) / ef1 \
              + (m1 - em1) * (m1 - em1) / em1
    return chisquare

def sim():
    # simulate females
    f0 = 0 # yes
    f1 = 0 # no
    for i in range(females_all):
        t = random.random()
        if t < p:
           f0 += 1
        else:
           f1 += 1
    # simulate males
    m0 = 0 # yes
    m1 = 1 # no
    for i in range(males_all):
        t = random.random()
        if t < p:
           m0 += 1
        else:
           m1 += 1
    return chisquare(f0, f1, m0, m1)

females_all = females_yes + females_no
males_all = males_yes + males_no
n = females_all + males_all  # all
p = (females_yes + males_yes) / n  # probability of yes for all
print "Facebook yes (all):", round(100 * p, 1), "%"
pf = females_yes / females_all
print "Facebook yes (females):", round(100 * pf, 1), "%"
pm = males_yes / males_all
print "Facebook yes (males:)", round(100 * pm, 1), "%"

makeGPanel(-1, 11, -250, 2750)
title("Chi-square test, use of Facebook")
drawGrid(0, 10, 0, 2500)
```

```
h = [0] * 101

repeat z:
    c = int(10 * sim())  # magnification factor of 10
    if c < 100:
        h[c] += 1
    else:
        h[100] += 1

showDistribution()
s = showLimit(0.95)

c = chisquare(females_yes, females_no, males_yes, males_no)
print "critical value:", s
print "observed:", c,
if c <= s:
   print "- the same behavior"
else:
   print "- not the same behavior"
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Le résultat est étonnant : le seuil de signification du χ^2 se trouve autour des 3.8 [**plus...**]. Les résultats de l'enquête donnent un χ^2 de 2.7. . On peut donc en conclure que, bien que la proportion de femmes inscrites sur Facebook soit bien plus élevée que pour les hommes, on ne peut pas prouver statistiquement à partir de cette enquête que leur comportement sur Facebook est significativement différent de celui des hommes.

EXERCICES

1. Pour tester l'efficacité d'un médicament de manière scientifique, on effectue une étude en double aveugle sur deux groupes de patients atteints d'une maladie donnée. On administre le médicament testé au premier groupe et un placebo au deuxième groupe. Après analyse des résultats du traitement, on obtient les données suivantes:

	Après traitement guéri	Après traitement malade	% de personnes guéries
Avec médicament	22	13	62.9 %
Avec placebo	11	17	39.3 %

Le pourcentage de personnes guéries dans le groupe traité au médicament est bien supérieur à celui du groupe traité avec le placebo. Ces résultats indiquent-ils de manière significative que le médicament est efficace?

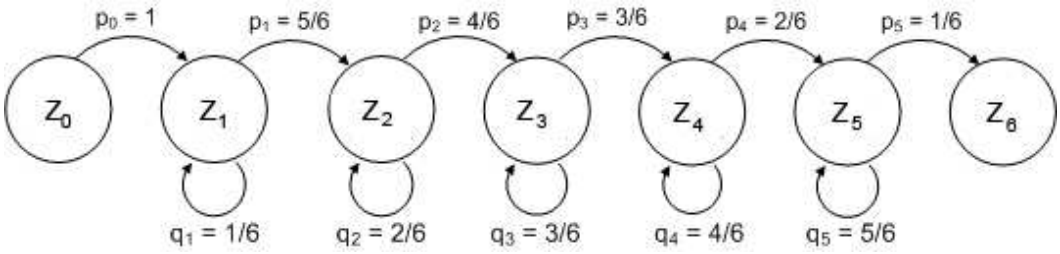
8.4 TEMPS MOYEN D'ATTENTE

INTRODUCTION

De nombreux systèmes présentent un comportement temporel qui peut être décrit par des transitions d'un état à l'autre. La transition d'un état système Z_i à un état suivant Z_k est déterminée par la probabilité pik. Dans l'exemple suivant, on jette un dé et l'on considère le nombre de faces du dé différentes déjà survenues comme une variable d'état:

Z_0: Encore aucun jet effectué
Z_1: Un jet effectué
Z_2: Deux faces différentes sont déjà survenues
etc.

On peut illustrer la transition entre les états dans le schéma suivant o **chaîne de Markov**



Les probabilités sont interprétées de la manière suivante : si l'on a déjà obtenu *n* faces différentes, la probabilité d'obtenir à nouveau une de ces faces vaut n/6 tandis que la probabilité d'obtenir une face encore jamais survenue vaut (6-n)/6. Vous pouvez essayer de déterminer combien de fois il faut relancer le dé en moyenne pour que toutes les faces apparaissent au moins une fois.

CONCEPTS DE PROGRAMMATION: *chaine de Markov, temps d'attente, paradoxe du temps d'attente.*

TEMPS MOYEN D'ATTENTE

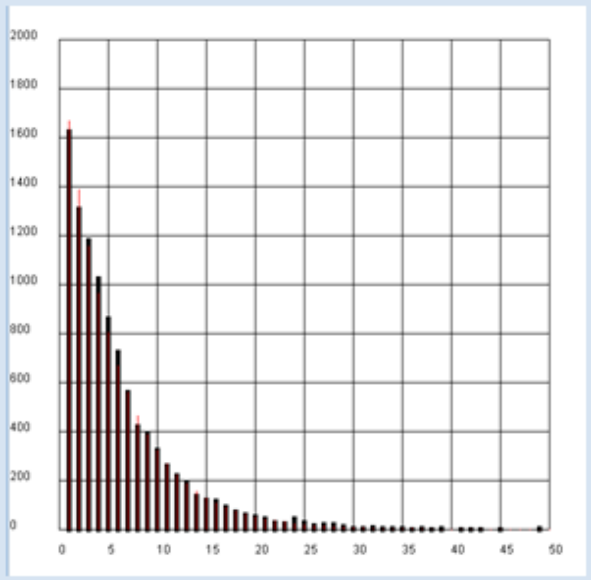
Si l'on jette le dé à intervalles de temps réguliers, la question précédente revient à déterminer le temps nécessaire pour que toutes les faces surviennent au moins une fois. On appelle ceci le **temps moyen d'attente**. On peut déterminer ce temps en faisant la réflexion suivante : le temps moyen pour passer de Z_0 à Z_6correspond à la somme des temps d'attente pour l'ensemble des transitions. Mais que vaut alors le temps d'attente de chaque transition individuelle?

Notre premier programme met en évidence la propriété la plus essentielle des problèmes de temps d'attente:

Si p est la probabilité de passer de Z_1 à Z_2le temps d'attente s'élève environ à u = 1/p (dans une unité appropriée).

Dans la simulation suivante, on cherche à déterminer le temps d'attente nécessaire pour obtenir une certaine face, par exemple un 6. Dans le cas présent, une étape de la simulation ne consiste pas en un unique jet de dé. En effet, on lance le dé aussi souvent que nécessaire jusqu'à l'obtention d'un 6 au sein de la fonction *sim()* qui retourne le nombre de jets qui ont été nécessaires. On répète l'expérience 10'000 fois et on calcule le nombre moyen de jets nécessaires.

Parallèlement, on affiche le nombre de jets nécessaires pour obtenir un 6 dans un diagramme de fréquences avec k = 1, 2, 3,... (on s'arrête à k = 50).




```
from gpanel import *
import random

n = 10000
p = 1/6

def sim():
    k = 1
    r = random.randint(1, 6)
    while r != 6:
        r = random.randint(1, 6)
        k += 1
    return k

makeGPanel(-5, 55, -200, 2200)
drawGrid(0, 50, 0, 2000)
title("Waiting on a 6")
h = [0] * 51
lineWidth(5)
sum = 0
repeat n:
    k = sim()
    sum += k
    if k <= 50:
        h[k] += 1
        line(k, 0, k, h[k])
mean_exp = sum / n

lineWidth(1)
setColor("red")
sum = 0
for k in range(1, 1000):
    pk = (1 - p)**(k - 1) * p
    nk = n * pk
    sum += nk * k
    if k <=50:
        line(k, 0, k, nk)
mean_theory = sum / n
title("Experiment: " + str(mean_exp) + "Theory: " + str(mean_theory))
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Le résultat est intuitivement évident : puisque la probabilité d'obtenir une des faces du dé vaut p=1/6, il faut en moyenne u = 1 / p = 6 jets pour obtenir cette face.

Il est également instructif d'afficher la valeur théorique des fréquences comme une ligne rouge. Pour ce faire, il faut tenir compte des considérations suivantes en ce qui concerne la probabilité d'obtenir un 6:

- Un 6 lors du premier jet: $p_1 = p$
- Pas de 6 dans le premier jet, mais 6 au second jet: $p_2 = (1 - p) * p$
- Pas de 6 lors des deux premiers jets, mais 6 lors du troisième: $p_3 = (1 - p) * (1 - p) * p$
- Pas de 6 lors des (k-1) premiers jets, mais 6 au *k*-ième jet:: $p_k = (1 - p)^{k-1} * p$

Pour obtenir les fréquences théoriques, on multiplie ces probabilités par le nombre *n* d'essais [**plus...**].

PROGRAMMER AU LIEU DE CALCULER

Pour résoudre le problème défini précédemment consistant à calculer le temps d'attente moyen jusqu'à ce que toutes les faces apparaissent au moins une fois, on décide d'adopter l'approche théorique. On interprète le processus comme une chaîne de Markov et l'on ajoute les temps d'attente de chaque transition individuelle:

u = 1 + 6/5 + 6/4 + 6/3 + 6/2 + 6 = 14.7

On pourrait aussi adopter une approche empirique consistant à écrire un simple programme permettant de déterminer ce nombre à l'aide d'une simulation. Pour ce faire, on procède toujours de la même manière : on définit une fonction *sim()* dans laquelle l'ordinateur recherche une seule solution en utilisant les nombres aléatoires. Cette fonction retourne le nombre d'étapes nécessaires. On répète ensuite cette tâche de nombreuses fois, disons 1'000 fois, et l'on détermine la valeur moyenne.

La fonction *sim()* utilise une liste *z* dans laquelle on insère les faces qui ne s'y trouvent pas déjà. Dès que cette liste possède six éléments, on sait que toutes les faces du dé sont survenues.

```
import random

n = 10000

def sim():
    z = []
    i = 0
    while True:
        r = random.randint(1, 6)
        i += 1
        if not r in z:
            z.append(r)
        if len(z) == 6:
            return i

sum = 0
repeat n:
    sum += sim()

print "Mean waiting time:", sum / n
```

MEMENTO

La simulation informatique livre la valeur 14,68 qui fluctue légèrement d'une fois à l'autre mais qui correspond à la prédiction théorique. On remarque donc que l'ordinateur peut être utilisé pour vérifier rapidement l'exactitude d'un résultat théorique.

Cependant, la détermination théorique du temps d'attente peut devenir très complexe pour des problèmes pourtant relativement simples. Si, par exemple, on essaie de déterminer le temps moyen d'attente jusqu'à l'obtention d'une certaine somme des nombres, le problème est extrêmement simple à résoudre à l'aide d'une simulation informatique.

```
import random

n = 10000
s = 7  # rolled sum of the die numbers

def sim():
    i = 0
    total =  0
    while True:
        i += 1
        r = random.randint(1, 6)
        total += r
        if total >= s:
            break
    return i

sum = 0
repeat n:
    sum += sim()

print "Mean waiting time:", sum / n
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

On obtient un temps moyen d'attente d'environ 2.52 jets pour obtenir une somme de 7. Ce résultat est assez surprenant puisque l'espérance pour chaque jet vaut 3.5. De ce fait, on peut admettre qu'il faut lancer le dé en moyenne deux fois pour obtenir une somme de points supérieure ou égale à 7. Le calcul théorique, au contraire de la simulation, peut facilement prendre des heures. Voici le résultat théorique : 117 577 / 46'656 = 2.5008.

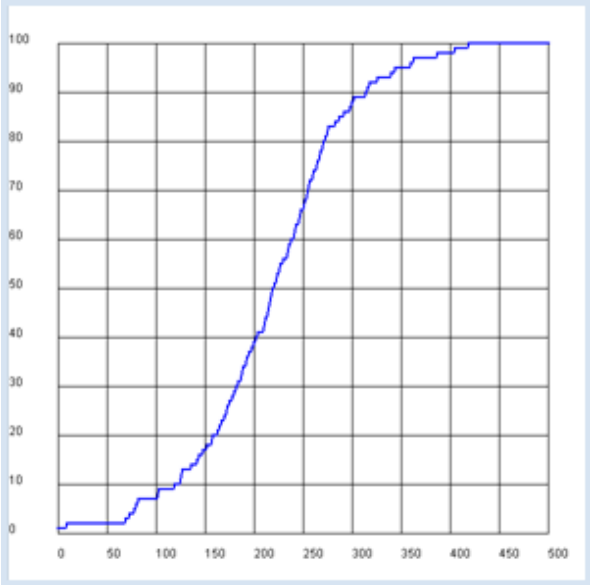
De ce fait, même les mathématiciens utilisent l'ordinateur pour éprouver rapidement un résultat théorique et vérifier leurs hypothèses.

PROPAGATION D'UNE MALADIE

Bien qu'elle soit fictive, admettons l'histoire suivante puisqu'elle comporte certains parallèles avec le vécu réel de certaines communautés d'êtres vivants:

"100 personnes vivent sur une île des Caraïbes coupée du reste du monde. Un vieillard est touché par une maladie qu'il a contractée en mangeant de manière inconsidérée un oiseau migrateur malade. Lorsqu'un individu malade rencontre un individu en bonne santé, ce dernier tombe malade rapidement. Toutes les deux heures, deux personnes se rencontrent au hasard."

On veut déterminer par une simulation informatique la manière dont la maladie se répand. Pour ce faire, on détermine le nombre de personnes infectées en fonction du temps.



Une bonne façon de représenter cette situation consiste à travailler avec une liste de valeurs booléennes où le caractère « en bonne santé » est codé par *False* et « malade » est codé par *True*. L'avantage de cette structure de données réside dans le fait qu'au sein de la fonction *pair()*, l'interaction entre deux personnes peut simplement être réalisée à l'aide de la fonction logique OU (or):

1. personne avant	**2. personne avant**	**Les deux personnes après**
saine (False)	saine (False)	saine (False)
saine (False)	malade (True)	maladel (True)
malade (True)	saine (False)	malade (True)
malade (True)	malade (True)	malade (True)

```
from gpanel import *
import random

def pair():
    # Select two distinct inhabitants
    a = random.randint(0, 99)
    b = a
    while b == a:
        b = random.randint(0, 99)
    z[a] = z[a] or z[b]
    z[b] = z[a]

def nbInfected():
    sum = 0
    for i in range(100):
        if z[i]:
            sum += 1
    return sum

makeGPanel(-50, 550, -10, 110)
title("The spread of an illness")
drawGrid(0, 500, 0, 100)
lineWidth(2)
setColor("blue")

z = [False] * 100
tmax =  500
t = 0
```

```
a = random.randint(0, 99)
z[a] = True # random infected inhabitant
move(t, 1)

while t <= tmax:
    pair()
    infects = nbInfected()
    t += 1
    draw(t, infects)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

On observe un comportement temporel dans lequel la propagation de la maladie est lente au début, puis rapide, puis à nouveau lente. Ce comportement s'explique par le fait qu'au début, la probabilité qu'une personne malade rencontre une personne en bonne santé est relativement faible. Par la suite, puisque de nombreuses personnes sont malades, cette probabilité augmente fortement ce qui produit une propagation rapide de la maladie. Dans la dernière phase, comme il n'y a presque plus que des personnes malades, la probabilité qu'une personne saine et une personne malade se rencontrent devient à nouveau faible. [**plus...**].

Une question intéressante consiste à savoir combien il faut de temps en moyenne pour que toute la population de l'île tombe malade. On peut répondre à cette question directement à l'aide d'une simulation informatique exécutée de nombreuses fois sur une même population. On procède en comptant le nombre d'étapes de simulation nécessaires pour que tout le monde tombe malade.

```
import random

n = 1000 # number experiment

def pair():
    # Select two distinct inhabitants
    a = random.randint(0, 99)
    b = a
    while b == a:
        b = random.randint(0, 99)
    z[a] = z[a] or z[b]
    z[b] = z[a]

def nbInfected():
    sum = 0
    for i in range(100):
        if z[i]:
            sum += 1
    return sum

def sim():
    global z
    z = [False] * 100
    t = 0
    a = random.randint(0, 99)
    z[a] = True # random infected inhabitant
    while True:
        pair()
        t += 1
        if nbInfected() == 100:
            return t

sum = 0
for i in range(n):
    u = sim()
    print "Experiment #", i + 1, "Waiting time:", u
```

```
    sum += u

print "Mean waiting time:", sum / n
```

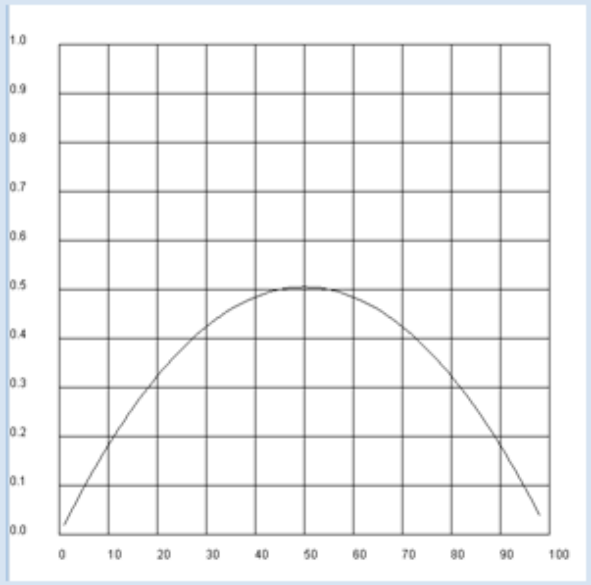
Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

On peut également illustrer la propagation de la maladie à l'aide d'une chaîne de Markov. Un état donné de la chaîne de Markov est caractérisé par le nombre de personnes infectées. Pour une population de 100 personnes, le temps nécessaire pour que tout le monde tombe malade correspond à la somme des temps d'attente pour chaque transition de *k* à *k+1* personnes malades, pour *k* allant de 1 à 99. De plus, pour réaliser cette simulation, il est nécessaire de connaître la probabilité de transition p_k.

La probabilité p_k est la somme des probabilités de choisir en premier une personne malade et ensuite une personne saine ou vice-versa :

$$p_k = \frac{k}{n} * \frac{n-k}{n-1} + \frac{n-k}{n} * \frac{k}{n-1} = 2 * \frac{k*(n-k)}{n*(n-1)}$$

Le programme représente également graphiquement les valeurs de p_k et détermine la somme des inverses de p_k.




```
from gpanel import *

n  = 100

def p(k):
    return 2 * k * (n - k) / n / (n - 1)

makeGPanel(-10, 110, -0.1, 1.1)
drawGrid(0, 100, 0, 1.0)

sum = 0
for k in range(1, n - 1):
    if k == 1:
        move(k, p(k))
    else:
        draw(k, p(k))
    sum += 1 / p(k)

title("Time until everyone is ill: " + str(sum))
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

En utilisant la théorie des chaines de Markov, on obtient donc un temps d'attente moyen de 463 heures jusqu'à ce que toutes les personnes de l'île soient infectées, ce qui correspond à environ 20 jours.

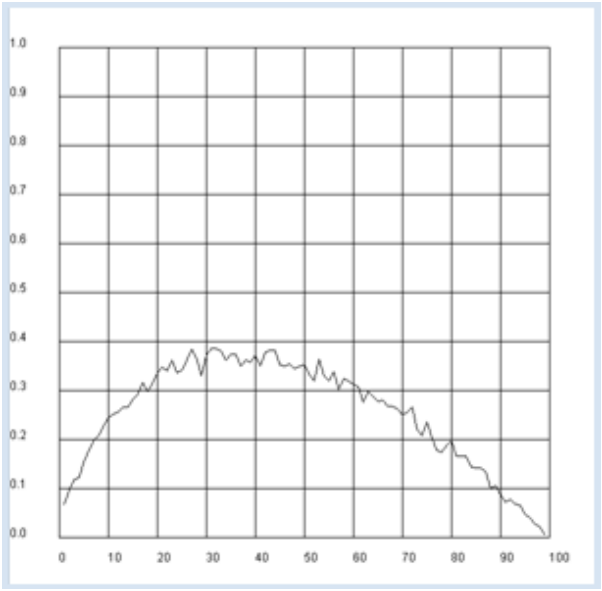
RECHERCHE DE PARTENAIRE À L'AIDE D'UN PROGRAMME

Une question intéressante en pratique concerne la stratégie optimale de recherche d'un partenaire de vie. On fait en l'occurrence l'hypothèse que cent partenaires possèdent des niveaux de qualification croissants. Ils seront présentés dans un ordre aléatoire et, dans une phase d'apprentissage, on a la possibilité de les ordonner correctement par niveau croissant de compétences en se basant sur les évaluations précédentes. Cependant, on ne connait pas le niveau de compétence maximal. Lors de chaque présentation, on doit décider si l'on accepte ou si l'on rejette le partenaire. Quelle est la meilleure stratégie pour s'assurer de choisir le meilleur partenaire avec une grande probabilité?

Pour cette simulation, on crée dans la fonction *sim(x)* une liste *t* de 100 niveaux de compétences de 0 à 99 ordonnés aléatoirement à l'aide de la fonction *shuffle()*.

Ensuite, on passe à la phase de sélection débutant par une phase d'apprentissage de longueur *x* fixe lors de laquelle on détermine l'indice du partenaire ayant le niveau de compétence le plus élevé.

On simule ensuite ce processus 1000 fois en utilisant une valeur de *x* bien précise et l'on détermine la probabilité de choisir le meilleur partenaire pour cette valeur de *x*. On représente ensuite cette probabilité graphiquement en fonction de la longueur *x* de la phase d'apprentissage.




```
import random
from gpanel import *

n = 1000 # Number of simulations
a = 100 # Number of partners

def sim(x):
    # Random permutation [0..99]
    t = [0] * 100
    for i in range(0, 100):
        t[i] = i
    random.shuffle(t)
    best = max(t[0:x])
    for i in range(x, 100):
        if  t[i] > best:
            return [i, t[i]]
    return [99, t[99]]

makeGPanel(-10, 110, -0.1, 1.1)
title("The probability of finding the best partner from  100")
drawGrid(0, 100, 0, 1.0)

for x in range(1, 100):
    sum = 0
    repeat n:
        z = sim(x)
        if z[1] == 99:  # best score
            sum += 1
    p = sum / n
    if x == 1:
        move(x, p)
    else:
```

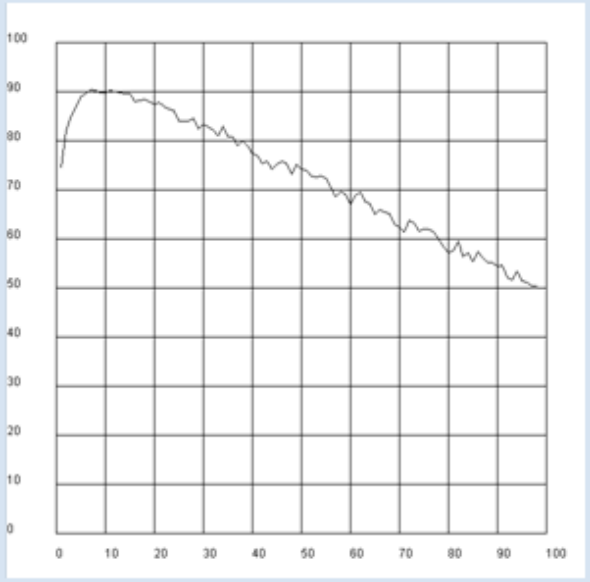
```
        draw(x, p)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Apparemment, on a les meilleures chances de choisir le meilleur partenaire après une phase d'apprentissage de longueur 37 [**plus...**].

On peut cependant aussi optimiser passablement la méthode d'échantillonnage en utilisant un critère d'optimalité qui ne recherche pas le meilleur partenaire absolu mais qui se contente d'un niveau aussi élevé que possible. Pour se faire, on examine à l'aide d'une simulation similaire le niveau de qualification moyen du partenaire sélectionné pour une longueur *x* de la phase d'apprentissage.




```
import random
from gpanel import *

n = 1000 # Number of simulations

def sim(x):
    # Random permutation [0..99]
    t = [0] * 100
    for i in range(0, 100):
        t[i] = i
    random.shuffle(t)
    best = max(t[0:x])
    for i in range(x, 100):
        if  t[i] > best:
            return [i, t[i]]
    return [99, t[99]]

makeGPanel(-10, 110, -10, 110)
title("Mean qualification after waiting for a  partner")
drawGrid(0, 100, 0, 100)

for x in range(1, 99):
    sum = 0
    repeat n:
        u = sim(x)
        sum += u[1]
    y = sum / n
    if x == 1:
       move(x, y)
    else:
       draw(x, y)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Ce deuxième critère d'optimalité livre un résultat complètement différent : il faut choisir le prochain partenaire ayant les meilleures compétences déjà après une phase d'apprentissage d'environ 10 présentations.

On peut également effectuer la simulation pour un nombre de partenaires plus réaliste et observer que la phase d'apprentissage optimale demeure relativement courte.

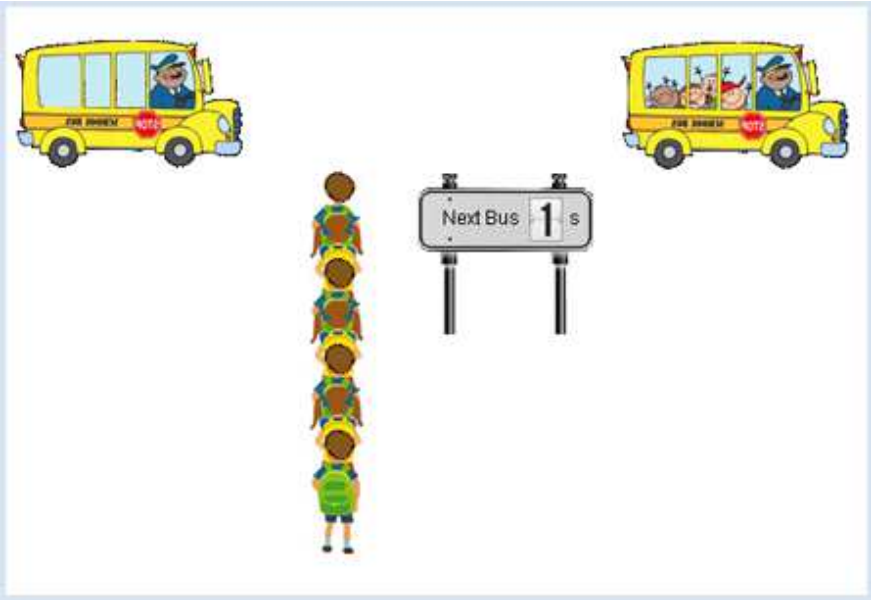
PARADOXE DU TEMPS D'ATTENTE

Attendre un transport public à l'arrêt de bus ou à la gare fait partie intégrante de notre vie quotidienne. Nous allons tenter de déterminer le temps moyen d'attente à l'arrêt de bus pour une personne qui s'y rend de manière aléatoire, à savoir sans connaître l'horaire. On suppose tout d'abord qu'un transport s'arrête à la station exactement toutes les 6 minutes.

Il est clair qu'il faudra parfois attendre juste quelques secondes et parfois le temps maximal de 6 minutes. Il faut donc en moyenne attendre environ 3 minutes. Qu'en est-il par contre si les bus n'arrivent pas à intervalles réguliers mais avec une distribution uniforme de temps compris entre 2 et 10 minutes?

Puisque les transports arrivent également toutes les 6 minutes en moyenne dans cette situation, on pourrait croire que le temps d'attente moyen est également de 3 minutes. Le résultat surprenant, et donc paradoxal, est que le temps d'attente est dans ce cas supérieure à 3 minutes.

Pour s'en convaincre, on utilise une simulation animée permettant de déterminer le temps moyen d'attente sous l'hypothèse que les bus arrivent à l'arrêt de bus espacés par un intervalle de temps uniformément distribué entre 2 et 10 minutes. Dans la simulation, les minutes seront des secondes pour accélérer le processus. On peut se servir de la bibliothèque de jeux *JGameGrid* puisqu'elle permet de modéliser facilement des objets tels que des bus et des passagers à l'aide de sprites.



Le code du programme requiert certainement quelques explications :

Puisque l'on a affaire à des objets de type bus et passagers, ont modélise ceci à l'aide des classes *Bus* et *Passenger*. Les bus sont créés au sein d'une boucle infinie à la fin de la partie principale du programme en accord avec les implications statistiques de notre hypothèse. Lorsque la fenêtre graphique est fermée, la boucle infinie se termine puisque la méthode *isDisposed() renvoie alors la valeur False,* ce qui a pour effet de terminer le programme.

Les passagers doivent être générés périodiquement et affichés dans la queue. Le meilleur moyen de réaliser ceci est de définir une classe *PassengerFactory* qui dérive de la classe *Actor*. Bien que cette dernière ne possède pas d'image de sprite, sa méthode *act()* peut être utilisées pour générer des passagers et les insérer dans la grille de jeu *GameGrid*. On peut changer la période avec laquelle les objets sont générés à l'aide du compteur de cycles *nbCycles* (le cycle de simulation est fixé à 50 ms).

On fait avancer le bus à l'aide de la méthode *act()* de la classe *Bus* et on vérifie s'il est arrivé à l'arrêt à l'aide de ses coordonnées *x*. Lorsque le bus parvient à l'arrêt, on invoque la méthode *board()* de la classe *PassengerFactory* qui a pour effet de supprimer de la queue les passagers en attente. Simultanément, on change l'image de sprite du bus avec *show(1)* et l'on affiche le nouveau temps d'attente jusqu'à l'arrivée du prochain bus sur le panneau. On utilise la variable booléenne *isBoarded* pour garantir que ces actions ne soient effectuées qu'une seule fois.

Le panneau (scoreboard), qui est une instance de la classe *InformationPanel* constitue un gadget supplémentaire permettant d'afficher le temps jusqu'à l'arrivée du prochain bus. L'écran du panneau d'affichage sera mis à jour dans la méthode *act()* en sélectionnant une des 10 images de sprite (*digit_0.png* à *digit_9.png)*grâce la fonction *show()*.

```
from gamegrid import *
import random
import time

min = 2
max = 10

def random_t():
    return min + (max - min) * random.random()

# ---------------- class PassengerFactory ----------
class PassengerFactory(Actor):
    def __init__(self):
        self.nbPassenger = 0

    def board(self):
        for passenger in getActors(Passenger):
            passenger.removeSelf()
            passenger.board()
        self.nbPassenger = 0

    def act(self):
        if self.nbCycles % 10 == 0:
            passenger = Passenger(random.randint(0, 1))
            addActor(passenger, Location(400, 120 + 27 * self.nbPassenger))
            self.nbPassenger += 1

# ---------------- class Passenger -----------------
class Passenger(Actor):
    totalTime = 0
    totalNumber = 0

    def __init__(self, i):
        Actor.__init__(self, "sprites/pupil_" + str(i) + ".png")
        self.createTime = time.clock()

    def board(self):
        self.waitTime = time.clock() - self.createTime
```

```
        Passenger.totalTime += self.waitTime
        Passenger.totalNumber += 1
        mean = Passenger.totalTime / Passenger.totalNumber
        setStatusText("Mean waiting time: " + str(round(mean, 2)) + " s")

# ---------------- class Car ------------------------
class Bus(Actor):
    def __init__(self, lag):
        Actor.__init__(self, "sprites/car1.gif")
        self.lag = lag
        self.isBoarded = False

    def act(self):
        self.move()
        if self.getX() > 320 and not self.isBoarded:
            passengerFactory.board()
            self.isBoarded = True
            infoPanel.setWaitingTime(self.lag)
        if self.getX() > 1650:
            self.removeSelf()

# ---------------- class InformationPanel ----------
class InformationPanel(Actor):
    def __init__(self, waitingTime):
        Actor.__init__(self, "sprites/digit.png", 10)
        self.waitingTime = waitingTime

    def setWaitingTime(self, waitingTime):
        self.waitingTime = waitingTime

    def act(self):
        self.show(int(self.waitingTime + 0.5))
        if self.waitingTime > 0:
            self.waitingTime -= 0.1

periodic = askYesNo("Departures every 6 s?")
makeGameGrid(800, 600, 1, None, None, False)
addStatusBar(20)
setStatusText("Acquiring data...")
setBgColor(Color.white)
setSimulationPeriod(50)
show()
doRun()
if periodic:
    setTitle("Warting Time Paradoxon - Departure every 6 s")
else:
    setTitle("Waiting Time Paradoxon - Departure between 2 s and 10 s")

passengerFactory = PassengerFactory()
addActor(passengerFactory, Location(0, 0))

addActor(Actor("sprites/panel.png"), Location(500, 120))
addActor(TextActor("Next Bus"), Location(460, 110))
addActor(TextActor("s"), Location(540, 110))
infoPanel = InformationPanel(4)
infoPanel.setSlowDown(2)
addActor(infoPanel, Location(525, 110))

while not isDisposed():
    if periodic:
        lag = 6
    else:
        lag = random_t()
    bus = Bus(lag)
    addActor(bus, Location(-100, 40))
    a = time.clock()
    while time.clock() - a < lag and not isDisposed():
        delay(10)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

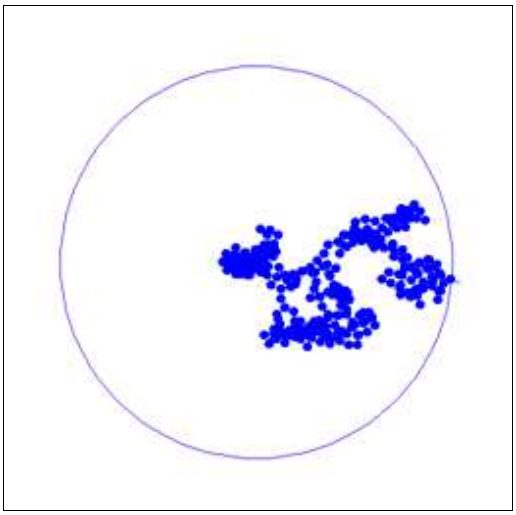
MEMENTO

La simulation montre bien que le temps moyen d'attente se situe aux alentours de 3.5 minutes, ce qui est clairement plus long que les 3 minutes d'attente nécessaires dans le cas où les bus sont tous espacés de 6 minutes exactement. On peut expliquer cette différence de la manière suivante : il est bien plus probable d'arriver à l'arrêt de bus lorsque le panneau affiche un temps d'attente compris entre 2 et 10 minutes que lorsqu'il affiche un temps compris entre 0 et 2 minutes. De ce fait, on a de fortes chances d'attendre plus que 3 minutes.

EXERCICES

1. Un enfant reçoit quotidiennement une pièce de 10, 20 ou 50 centimes avec une probabilité égale de 1/3. Combien de jours doit-il attendre en moyenne pour pouvoir s'acheter un livre qui coûte 10 francs?

2. Une personne perdue se déplace à partir du milieu de la fenêtre en faisant des pas de 10 pixels dans une direction aléatoire (marche aléatoire = random walk). Quel est le temps moyen *u* jusqu'à ce que cette personne se trouve pour la première fois éloignée d'une distance *r* de son point de départ ? Simuler ce mouvement à l'aide d'une tortue (cachée) pour les valeurs r = 100, 200, 300. D'après vous, quelle est la relation entre *r* et *u*?



3. Modifier le programme de simulation du paradoxe de temps d'attente à l'arrêt de bus de telle sorte que les bus arrivent soit après 2 secondes soit après 10 secondes avec une égale probabilité de ½. Déterminer le temps moyen d'attente dans ce cas de figure.

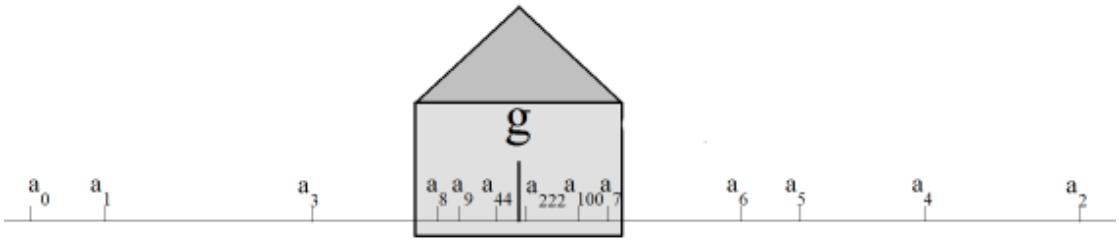
8.5 SUITES ET CONVERGENCE

INTRODUCTION

Les suites de nombres passionnent l'humanité depuis la nuit des temps. Dans l'Antiquité déjà, aux alentours de 200 avant J.C., Archimède était parvenu à approximer la valeur du nombre Pi. Il utilisa une suite de nombres correspondant au périmètre de polygones réguliers à *n* côtés inscrits dans un cercle de rayon *r* fixe, en prenant un nombre *n* de côtés toujours plus grand. Il lui est apparu clairement qu'il était fructueux de concevoir le cercle comme un cas limite de polygone possédant un nombre « infini » de côtés. Il en a conclu que la suite des périmètres de ces cercles tendait vers le périmètre du cercle inscrit.

Une suite de nombres est constituée des nombres a_0, a_1, a_2, etc.,à savoir des termes an d'indices n = 0, 1, 2, etc. (les suites commencent parfois à n = 1). Une règle de formation détermine clairement de manière unique la valeur de chacun des termes. Si ak est déterminé pour n'importe quel nombre naturel *k* aussi grand que l'on veut, on dit que la suite est infinie. Les termes de la suite peuvent être déterminés à l'aide d'une formule explicite dépendant de *n*. Cependant, le terme de rang *n* peut également être calculé à partir des termes précédents à l'aide d'une formule de récurrence. Dans ce cas, il est également nécessaire de connaitre la valeur des termes de départ.

Une suite convergente possède une propriété très particulière : il existe un unique nombre *g* appelé **limite** vers lequel tendent les termes de la suite. Cela veut dire que l'on peut choisir un voisinage aussi petit que l'on veut autour de la limite *g* de sorte qu'à partir d'un rang *n* de la suite, aucun terme suivant ne se trouvera en dehors de ce voisinage. On peut se représenter ce voisinage comme une maison aux alentours de la limite : avant ce *n* donné, les termes peuvent osciller autour de la maison (le voisinage) mais à partir d'un certain rang *n*, tous les termes de la suite se trouvent à l'intérieur de cette maison, aussi petite soit-elle.



Il est possible d'étudier les suites de nombres de manière expérimentale à l'aide de l'ordinateur. On peut les représenter graphiquement de différentes manières : on peut, à la manière de l'illustration ci-dessus, représenter tous les termes de la suite comme des points ou des petits traits et déterminer s'il elle converge vers **un point limite**. Il est également possible d'étudier le comportement de la suite **pour de grandes valeurs de *n*** en la représentant dans un graphique à deux dimensions où *n* se trouve sur l'axe horizontal et les termes an sur l'axe vertical.

CONCEPTS DE PROGRAMMATION: *Point limite, convergence, diagramme de bifurcation de Feigenbaum , chaos*

LE CHASSEUR ET SON CHIEN

Un chasseur marche avec son chien en direction de leur pavillon de chasse situé à une distance *d = 1000 m* à une vitesse de *1 m/s*. Comme le chien court plus vite que son maître, ils se déplacent de la manière suivante : le chien court seul à une vitesse *u = 20 m/s* en direction du pavillon de chasse, y fait demi-tour et s'en retourne vers son maître à la même vitesse. Dès qu'il atteint son

maître, il fait à nouveau demi-tour et cours à nouveau en direction du pavillon de chasse. Il poursuit ce comportement jusqu'à ce que les deux aient atteint le pavillon de chasse.

On aimerait simuler cette procédure à l'aide d'un programme. Lors de chaque clic sur le bouton dans le programme, on dessine le prochain point de rencontre entre le chasseur et son chien en écrivant la position correspondante à droite du point. Les positions successives de nombres forment une suite dont on aimerait étudier le comportement. Puisqu'il s'écoule un temps identique pour le chien et son chasseur entre chaque rencontre, l'augmentation de la position du chasseur représentée par les points peut être décrite de la manière suivante :

$$\frac{da}{u} = \frac{2 * (d - a) - da}{v}$$

On peut facilement en déduire la relation suivante avec des connaissances d'algèbre limitées:

$$da = c * (d - x) \qquad \text{avec} \qquad c = \frac{2 * u}{u + v}$$

Comme vous vous y attendiez certainement, les nombres an s'empilent jusqu'à atteindre le nombre limite 1000.

```
from gpanel import *

u = 1 # m/s
v = 20 # m/s
d = 1000  # m
a = 0     # hunter
h = 0     # dog
c = 2 * u / (u + v)
it = 0

makeGPanel(-50, 50, -100, 1100)
title("Hunter-Dog problem")
line(0, 0, 0, 1000)
line(-5, 1000, 5, 1000)
line(-5, 1050, 5, 1050)
line(-5, 1000, -5, 1050)
line(5, 1000, 5, 1050)

while not isDisposed():
    move(0, a)
    fillCircle(1)
    text(5, a, str(int(a)))
    getKeyWait()
    da = c * (d - a)
    dh = 2 * (d  - a) - da
    h += dh
    a += da
    it += 1
    title("it = " + str(it) + "; hunter = " + str(a) +
          " m; dog = " + str(h) + " m")
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

On dit que la suite de nombres an **converge** et que sa valeur limite est 1000.

Essayez de saisir la nature purement théorique de cette formulation du problème. Elle correspond à l'anecdote antique rapportant que Achille fut invité à faire la course avec une tortue en lui laissant 10 mètres d'avance puisqu'il courait 10 fois plus vite qu'elle. On raconte qu'Achille refusa la compétition car, selon lui, il n'avait aucune chance de la rattraper. Son argument était le suivant : pendant qu'il parcourrait les 10 premiers mètres, la tortue aurait avancé d'un mètre et se trouverait donc au onzième mètre. Pendant qu'il parcourrait le mètre suivant, la tortue aurait parcouru encore 10 cm et se trouverait donc à 11,1 m du départ. Pendant qu'il parcourrait les 10 centimètres suivants, la tortue aurait parcouru encore 1 cm et se trouverait donc à 11,11 m du départ et ainsi de suite. Que pensez-vous de ce raisonnement ?

DIAGRAMME DE BIFURCATION DE FEIGENBAUM

Nous avons déjà étudié la croissance logistique dans le cadre de la dynamique des populations. La taille de la population *xnew* dans la génération suivante est calculée à partir de sa taille actuelle x à l'aide d'une relation quadratique. Dans la suite, cette relation sera simplifiée par la relation

$$xnew = r * x * (1 - x)$$

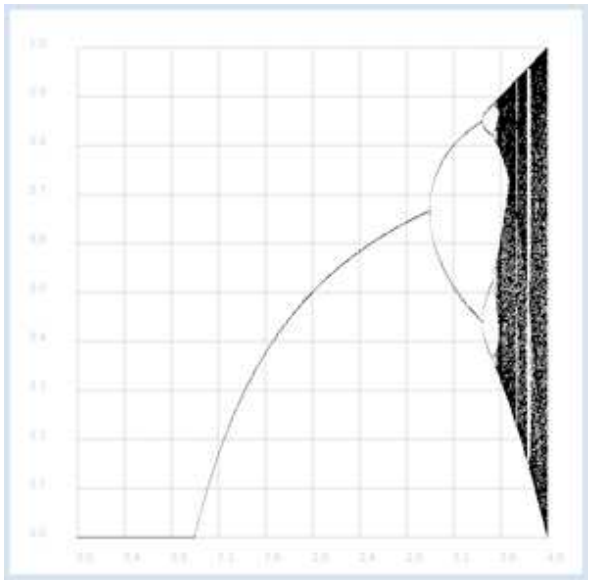
où le paramètre *r* peut être choisi arbitrairement. On voudrait savoir si la suite de nombres récursive

$$a_{n+1} = r * a_n * (1 - a_n)$$

qui en résulte avec $a_0 = 0.5$ converge et qu'elle serait alors sa valeur limite.

On étudie le comportement de cette suite à l'aide d'un programme extrêmement simple qui représente graphiquement, sous forme de points, les 1000 premiers termes de la suite pour 1000 valeurs équidistantes de *r* comprises entre 0 et 4.

Pour une valeur de *r* fixée, on commence toujours avec le même terme initial $a_0 = 0.5$ On ne dessine que les termes à partir du rang n = 500 puisque la seule chose qui nous intéresse est de savoir si la suite est convergente ou, au contraire, divergente.



```
from gpanel import *

def f(x, r):
     return r * x * (1 - x)

makeGPanel(-0.6, 4.4, -0.1, 1.1)
title("Tree Diagram")
drawGrid(0, 4.0, 0, 1.0, "gray")
for z in range(1001):
    r = 4 * z / 1000
```

```
    a = 0.5
    for i in range(1001):
        a = f(a, r)
        if i > 500:
            point(r, a)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

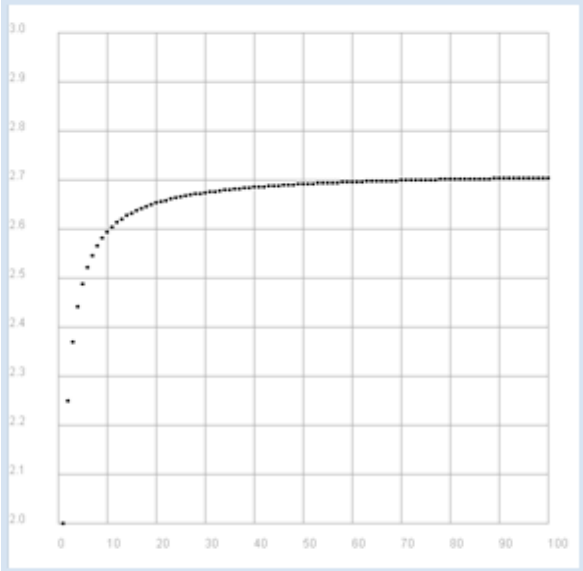
Dans cette expérience, on met en évidence les points d'accumulation de la suite pour un certain *r* donné. Sur la base de cette simulation informatique, on peut établir les hypothèses suivantes : pour r < 1 il y a un point d'accumulation en zéro et la suite converge vers 0. Pour des valeurs de *r* comprises entre *1* et *3*, la suite converge également. Pour des valeurs de *r* encore plus grandes, il y a d'abord deux puis plusieurs points d'accumulation. À partir d'une certaine valeur de r, la suite saute dans tous les sens de manière chaotique.

LE NOMBRE D'EULER

L'une des suites les plus célèbres est définie par le terme général suivant :

$$a_n = (1 + \frac{1}{n})^n \quad avec\ n = 1, 2, 3, \ldots$$

Il est très difficile de déterminer de manière intuitive le comportement de cette suite pour des valeurs croissantes de *n*. En effet l'expression *1 + 1/n* tend d'une part vers *1* lorsque *n* tend vers l'infini mais, de l'autre, on élève ce nombre à un exposant *n* toujours plus grand. Vous pouvez tenter d'élucider ce mystère à l'aide d'une expérience informatique.




```
from gpanel import *

def a(n):
     return (1 + 1/n)**n

makeGPanel(-10, 110, 1.9, 3.1)
title("Euler Number")
drawGrid(0, 100, 2.0, 3.0, "gray")
for n in range(1, 101):
    move(n, a(n))
    fillCircle(0.5)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

$a_n = (1 + \frac{1}{n})^n$ La suite converge vers un nombre de l'ordre de 2.7.

Il s'agit du **nombre** d'Euler, l'un des nombres les plus célèbres.

SUITES RAPIDEMENT CONVERGENTES POUR LE CALCUL DE PI

Le calcul d'un nombre quelconque de décimales de π a toujours constitué un défi pour les mathématiciens. Ce n'est qu'en 1995 que les mathématiciens Bailey, Borwein et Plouffe ont prouvé la formule BBP sous forme de somme permettant ce calcul. Ils ont montré que l'on peut obtenir exactement le nombre π comme limite d'une suite dont le terme de rang *n* est donné par la somme des

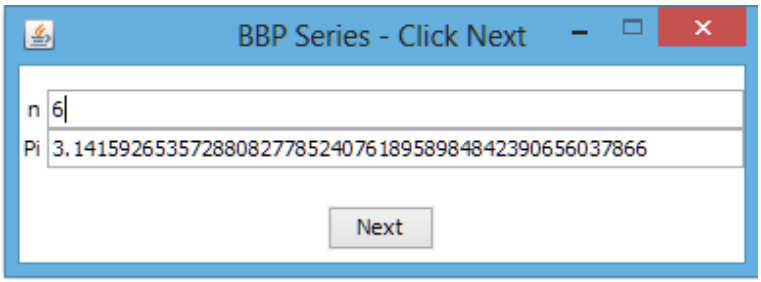
$$\frac{1}{16^k}\left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6}\right) .$$

pour *k* allant de *0* à *n.*

Le programme suivant utilise le module **decimal** qui permet de calculer avec des nombres décimaux avec une très grande précision. Le constructeur de cette classe crée un tel nombre à partir d'un entier ou d'en flottant et autorise les opérations arithmétiques habituelles de manière transparente.

La précision du calcul peut être réglée à l'aide de *getcontext().prec* et correspond en gros au nombre de décimales significatives.

À chaque pression d'une touche du clavier, le programme calcule le prochain terme de la suite qu'il affiche ensuite dans une boîte de dialogue *EntryDialog*.




```
from entrydialog import *
from decimal import *
getcontext().prec = 50

def a(k):
    return 1/16**Decimal(k) * (4 / (8 * Decimal(k) + 1) - 2
    / (8 * Decimal(k) + 4) - 1
    / (8 * Decimal(k) + 5) - 1 / (8 * Decimal(k) + 6))

inp = IntEntry("n", 0)
out = StringEntry("Pi")
pane0 = EntryPane(inp, out)
btn = ButtonEntry("Next")
pane1 = EntryPane(btn)
dlg = EntryDialog(pane0, pane1)
dlg.setTitle("BBP Series - Click Next")
n = 0
s = a(0)
out.setValue(str(s))
while not dlg.isDisposed():
    if btn.isTouched():
        n = inp.getValue()
        if n == None:
            out.setValue("Illegal entry")
        else:
            n += 1
            s += a(n)
            inp.setValue(n)
            out.setValue(str(s))
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Après 40 itérations, la valeur de π affichée ne change plus.

Le programme se termine lorsque l'on ferme la fenêtre puisque *isDisposed()* est *True*.

EXERCICES

1. La suite de Fibonacci est définie de la manière suivante : chaque terme est la somme des deux termes précédents. Les deux premiers termes de la suite valent 1. Calculer les 30 premiers termes de cette suite et les afficher sur un graphe x-y.

2. La suite de Fibonacci diverge alors que la suite des quotients de deux termes consécutifs converge. De manière similaire à l'exercice 1, représenter graphiquement cette suite de quotients et déterminer de manière approximative sa valeur limite.

3. Considérons la suite suivante de valeur initiale $a_0 = 1$:

$$a_{n+1} = \frac{1}{2} * \left(a_n + \frac{2}{a_n} \right)^n$$

Représenter les termes de cette suite comme des points sur la droite des nombres et afficher leur valeur dans la console. Comme vous pouvez le constater, cette suite est manifestement convergente. On peut estimer grossièrement sa valeur limite *x* de la manière suivante : pour de grandes valeurs de *n*, les termes de la suite sont si proches qu'il est impossible de les distinguer sur le graphique. On a donc, dans le cas limite, que

$$x = \frac{1}{2} * \left(x + \frac{2}{x} \right) \quad \text{à savoir} \quad x = \sqrt{2}$$

Expliquer de quelle manière ce résultat permet de calculer approximativement la racine carrée de 2 en n'utilisant que les 4 opérations arithmétiques de base. Expliquer comment on peut adapter cet algorithme pour calculer la racine carrée de n'importe quel nombre *z* positif.

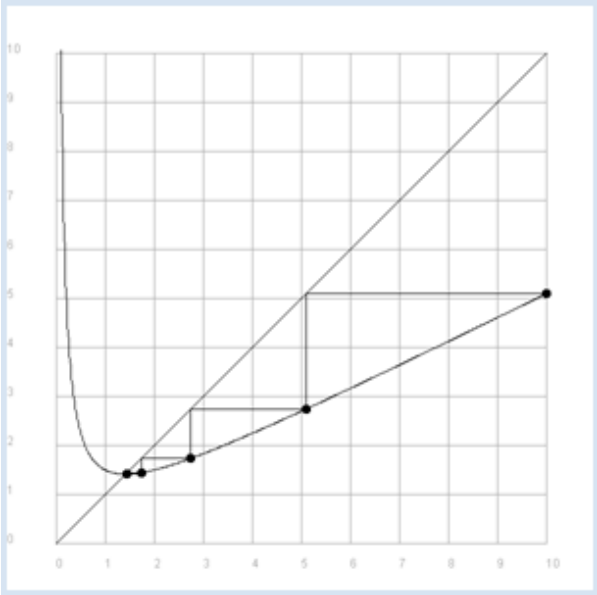
MATÉRIEL SUPPLÉMENTAIRE

RÉSOLUTION ITÉRATIVE D'UNE ÉQUATION

Comme vous avez pu le constater dans l'exercice 3, $x = \sqrt{2}$ est la solution de l'équation

$$x = f(x) \quad avec \quad f(x) = \frac{1}{2} * \left(x + \frac{2}{x} \right)$$

Il est très instructif de représenter graphiquement la procédure de résolution de cette équation. Pour ce faire, on dessine dans le même repère cartésien le graphe de la fonction $y = f(x)$ ainsi que la bissectrice du premier quadrant, savoir la droite d'équation cartésienne $y = x$. La solution se trouve à l'intersection de ces deux courbes.



La résolution itérative d'une équation correspond au passage successif d'un point du graphe de la fonction vers le point de la bissectrice de même ordonnée (déplacement horizontal) suivi du passage de ce point de la bissectrice vers le point du graphe de la fonction de même abscisse (déplacement vertical).

.

```
from gpanel import *

def f(x):
    return 1 / 2 * (x + 2 / x)

makeGPanel(-1, 11, -1, 11)
title("Iterative square root begins at x = 10. Press a  key...")
drawGrid(0, 10, 0, 10, "gray")

for i in range(10, 1001):
    x = 10 / 1000 * i
    if i == 10:
        move(x, f(x))
    else:
        draw(x, f(x))

line(0, 0, 10, 10)

x = 10
move(x, f(x))
fillCircle(0.1)
it = 0
while not isDisposed():
    getKeyWait()
    it += 1
    xnew = f(x)
    line(x, f(x), xnew, f(x))
    line(xnew, f(x), xnew, f(xnew))
    x = xnew
    move(x, f(x))
    fillCircle(0.1)
    title("Iteration " + str(it) + ": x = " + str(x))
```

MEMENTO

Cette expérience montre clairement qu'en seulement quelques itérations (pression sur une touche du clavier), les points convergent très rapidement vers le point d'intersection, ce qui permet d'obtenir très rapidement une précision de 10 décimales.

8.6 CORRÉLATION, RÉGRESSION

INTRODUCTION

Aussi bien en sciences naturelles que dans la vie de tous les jours, les collections de données sous forme de mesures jouent un rôle important. Il n'est cependant pas toujours nécessaire d'avoir recours à un instrument de mesure. Les données peuvent en effet également provenir d'un sondage lié à une étude statistique. Suite à une collecte de données, il est très important **d'interpréter** les données collectées. L'interprétation des données peut conduire à une affirmation qualitative, par exemple que les valeurs augmentent ou diminuent avec le temps. Elle peut également permettre de vérifier expérimentalement un résultat scientifique théorique.

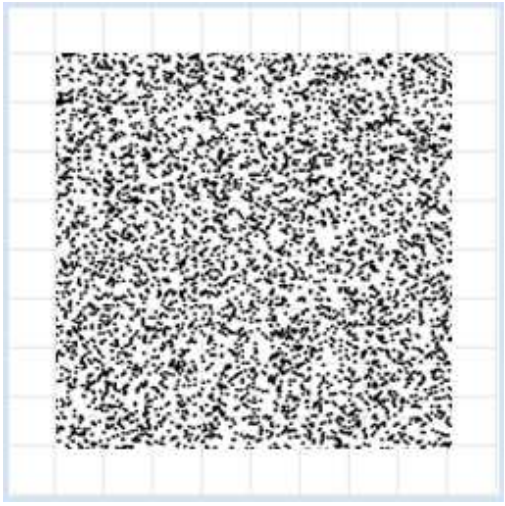
Une des difficultés majeures réside dans le fait que les mesures sont toujours sujettes à certaines fluctuations et ne sont jamais parfaitement reproductibles. Malgré ces incertitudes inhérentes aux mesures, le responsable d'une telle étude doit tout de même parvenir à des conclusions scientifiquement correctes. Les incertitudes de mesure ne proviennent pas nécessairement des instruments de mesure eux-mêmes car elles peuvent également provenir de la nature même de l'expérience. Par exemple, la « mesure » des résultats obtenus lors de jets de dés conduit à des valeurs dispersées entre 1 et 6. De ce fait, pour tout type de mesures, les considérations statistiques jouent un rôle central.

Très souvent, des variables *x* et *y* sont mesurées de pair dans une série de mesures et l'on se pose la question de savoir si elles sont corrélées (reliées) et si ces mesures présentent une certaine régularité. On appelle **visualisation de données** le fait de représenter les couples de mesures *(x, y)* comme des points dans un repère cartésien. On peut presque toujours reconnaître, à la simple vue de ce graphique, si les variables *x* et *y* sont dépendantes l'une de l'eau : il suffit d'observer à ce dessin la **distribution des valeurs mesurées**.

CONCEPTS DE PROGRAMMATION: *Visualisation de données, distribution de la grandeur mesurée, nuage de points, bruit, covariance, coefficient de corrélation, régression*

VISUALISATION DE DONNÉES DÉPENDANTES ET INDÉPENDANTES

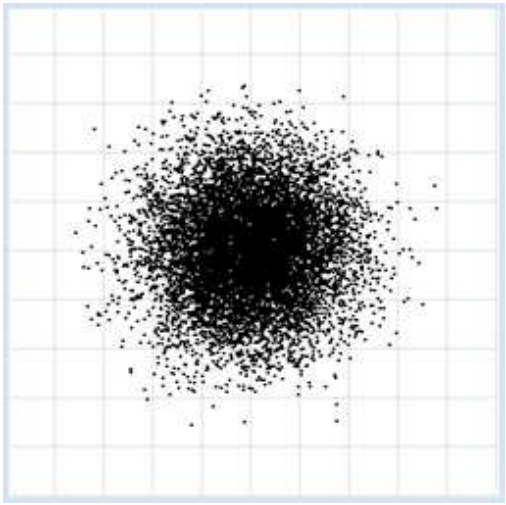
On peut aisément se représenter des données indépendantes dans un diagramme x-y en prenant pour *x* et *y* des nombres aléatoires uniformément distribués. Bien que cela ne soit pas nécessaire dans ce cas précis, on copie les valeurs mesurées dans les listes *xval* et *yval* avant de les représenter comme des points (x,y).



```
import random
from gpanel import *

z = 10000

makeGPanel(-1, 11, -1, 11)
title("Uuniformly distributed value pairs")
drawGrid(10, 10, "gray")

xval = [0] * z
yval = [0] * z

for i in range(z):
```

```
    xval[i] = 10 * random.random()
    yval[i] = 10 * random.random()
    move(xval[i], yval[i])
    fillCircle(0.03)
```

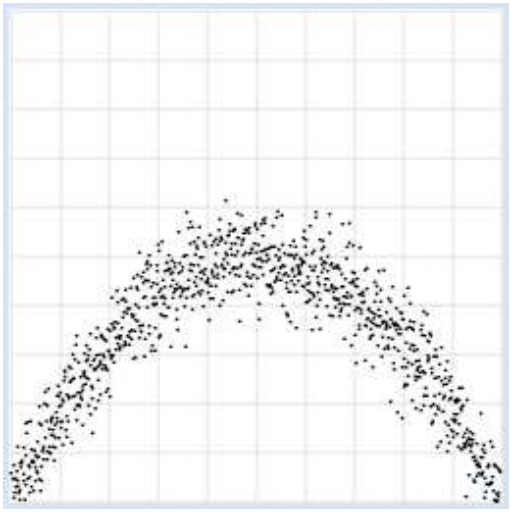
On obtient des graphiques en forme de galaxies lorsque les valeurs *x* et *y* sont indépendantes mais distribuées selon une loi normale autour d'une valeur moyenne. Dans l'exemple ci-contre, on a pris 5 comme valeur moyenne et 1 comme paramètre de dispersion. On appelle ce genre de graphique un **nuage de points**.



Il est très facile d'obtenir une série de nombres aléatoires suivant une loi normale (également appelée *gaussienne*) en Python avec la fonction *random.gauss(valeur_moyenne, dispersion)*.

```
import random
from gpanel import *

z = 10000

makeGPanel(-1, 11, -1, 11)
title("Normally distributed value pairs")
drawGrid(10, 10, "gray")

xval = [0] * z
yval = [0] * z

for i in range(z):
    xval[i] = random.gauss(5, 1)
    yval[i] = random.gauss(5, 1)
    move(xval[i], yval[i])
    fillCircle(0.03)
```

On peut facilement simuler une relation de dépendance entre deux variables *x* et *y*. Pour ce faire, prenons pour *x* une série de valeurs équidistantes en parcourant un intervalle par incréments réguliers et, pour chaque *x*, calculons *y* en fonction de *x* en ajoutant une petite fluctuation statistique. En physique, de telles fluctuations sont appelées **bruit.**



Le programme suivant représente le nuage de points pour *x = 0..10* avec un incrément de *0.01* et la fonction y = -x * (0.2 * x - 2) avec un bruit suivant une loi normale.

```
import random
from gpanel import *
import math

z = 1000
a = 0.2
```

```
b = 2

def f(x):
    y = -x * (a * x - b)
    return y

makeGPanel(-1, 11, -1, 11)
title("y = -x * (0.2 * x - 2) with normally distributed noise")
drawGrid(0, 10, 0, 10, "gray")

xval = [0] * z
yval = [0] * z

for i in range(z):
    x = i / 100
    xval[i] = x
    yval[i] = f(x) + random.gauss(0, 0.5)
    move(xval[i], yval[i])
    fillCircle(0.03)
```

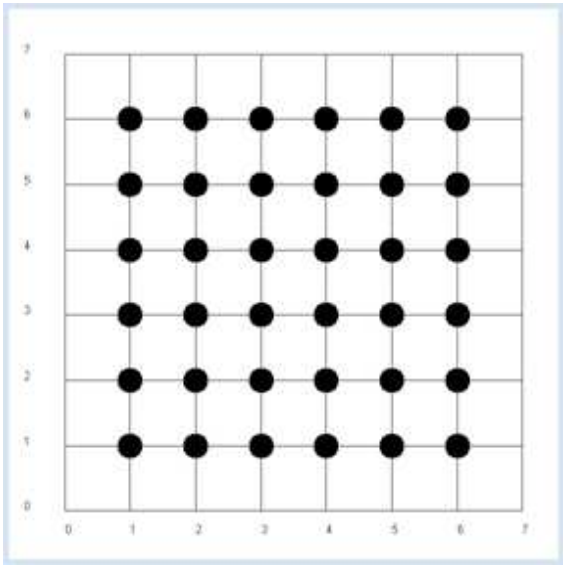
MEMENTO

On peut facilement reconnaître des relations de dépendance entre variables aléatoires grâce à la visualisation de données [**plus...**].

Avec une série de nombres aléatoires distribués de manière uniforme sur un intervalle [a, b], les nombres apparaissent avec la même fréquence dans chaque sous-intervalle de même longueur. Une distribution normale de nombres aléatoires (ou gaussienne) présente une forme de cloche et présente la particularité que 68% des valeurs se trouvent dans l'intervalle [moyenne – dispersion ; moyenne + dispersion].

LA COVARIANCE COMME MESURE DE LA DÉPENDANCE DE DEUX VARIABLES

En plus de vouloir rendre visible la dépendance entre deux variables grâce à un graphique, on veut pouvoir mesurer ce degré de dépendance par un nombre. Partons d'un exemple concret en considérant un double jet de dé où *x* est le résultat du premier jet et *y* le résultat du second. Effectuons cette expérience de nombreuses fois en représentant les paires de valeur dans un nuage de points. Puisque les deux valeurs *x* et *y* sont indépendantes, on obtient un nuage de points régulier. Si l'on calcule l'espérance de *x* et *y*, on obtient le résultat bien connu de 3.5.



Puisque les variables *x* et *y* sont indépendantes, la probabilité pxy d'obtenir la paire (x, y) lors d'un double jet est donnée par $p_{xy} = p_x * p_y$. Il n'y a qu'un pas pour faire l'hypothèse que cette **règle du produit des probabilités** est valable de manière générale lorsque les variables *x* et *y* sont indépendantes.

Lorsque les variables *x* et *y* sont indépendantes, l'espérance mathématique du produit *x*y* est donnée par le produit des espérances de *x* et *y* [**plus...**] .La simulation suivante confirme cette hypothèse :

```
from random import randint
from gpanel import *

z = 10000 # number of double rolls

def dec2(x):
    return str(round(x, 2))

def mean(xval):
    n = len(xval)
    sum = 0
    for i in range(n):
        sum += xval[i]
    return sum / n

makeGPanel(-1, 8, -1, 8)
title("Double rolls. Independent random variables")
addStatusBar(30)
drawGrid(0, 7, 0, 7, 7, 7, "gray")

xval =  [0] * z
yval =  [0] * z
xyval = [0] * z

for i in range(z):
    a = randint(1, 6)
    b = randint(1, 6)
    xval[i] = a
    yval[i] = b
    xyval[i] = xval[i] * yval[i]
    move(xval[i], yval[i])
    fillCircle(0.2)

xm = mean(xval)
ym = mean(yval)
xym = mean(xyval)
setStatusText("E(x) = " + dec2(xm) + \
          ",  E(y) = " + dec2(ym) + \
          ",  E(x, y) = " + dec2(xym))
```

MEMENTO

Il est conseillé d'utiliser la barre d'état pour écrire le résultat de la simulation. La fonction *dec2()* arrondit le résultat à deux chiffres et le retourne sous forme de chaîne de caractères.

Les valeurs sont bien entendu sujettes aux fluctuations statistiques.

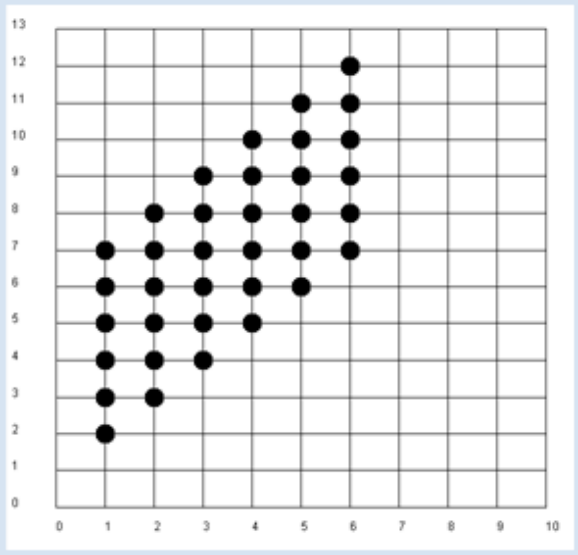
Dans la prochaine simulation, nous continuons d'examiner des paires de valeurs *(x,y)* lors de deux jets successifs d'un dé mais cette fois-ci, *x* représentera le résultat obtenu lors du premier jet et *y* la somme des deux jets successifs. Dans cette nouvelle situation, il est clair que *y* est dépendante de *x* puisque, par exemple, si *x=1*, la probabilité d'obtenir 4 pour *y* n'est pas la même que si *x=2*.

La simulation suivante confirme que la règle du produit ne s'applique pas lorsque les variables *x* et *y* sont dépendantes. Il est de ce fait raisonnable d'introduire l'écart avec la règle du produit, à savoir la différence

$$c = E(x*y) - E(x)*E(y)$$

comme une mesure de la dépendance entre *x* et *y*. On appelle cette grandeur la **covariance**.

En même temps, on voit dans le programme que la covariance peut également être calculée comme la somme des carrés des écarts avec la moyenne.




```
from random import randint
from gpanel import *

z = 10000 # number of double rolls

def dec2(x):
    return str(round(x, 2))

def mean(xval):
    n = len(xval)
    sum = 0
    for i in range(n):
        sum += xval[i]
    return sum / n

def covariance(xval, yval):
    n = len(xval)
    xm = mean(xval)
    ym = mean(yval)
    cxy = 0
    for i in range(n):
       cxy += (xval[i] - xm) * (yval[i] - ym)
    return cxy / n

makeGPanel(-1, 11, -2, 14)
title("Double rolls. Independent random variables")
addStatusBar(30)
drawGrid(0, 10, 0, 13, 10, 13, "gray")

xval =  [0] * z
yval =  [0] * z
xyval = [0] * z

for i in range(z):
    a = randint(1, 6)
    b = randint(1, 6)
    xval[i] = a
    yval[i] = a + b
    xyval[i] = xval[i] * yval[i]
    move(xval[i], yval[i])
    fillCircle(0.2)

xm = mean(xval)
ym = mean(yval)
xym = mean(xyval)
c = xym - xm * ym
setStatusText("E(x) = " + dec2(xm) + \
          ",  E(y) = " + dec2(ym) + \
```

```
          ",  E(x, y) = " + dec2(xym) + \
          ",  c = " + dec2(c) + \
          ",  covariance = " + dec2(covariance(xval, yval)))
```

MEMENTO

On obtient environ 2,9 pour la covariance des valeurs obtenues dans cette simulation. La covariance est une bonne mesure de la dépendance entre deux variables aléatoires.

LE COEFFICIENT DE CORRÉLATION

La covariance que nous venons d'introduire présente également un désavantage. Selon l'échelle des valeurs de *x* et *y* mesurées, la covariance livre des grandeurs différentes même si les grandeurs présentent le même degré de dépendance. On peut le voir facilement si l'on refait la même expérience avec des dés dont les faces valent 10, 20, 30, 40, 50, 60 au lieu de 1, 2, 3, 4, 5, 6. La covariance va alors beaucoup changer bien que le degré de dépendance entre *x* et *y* soit dans ce cas exactement le même qu'avec un dé habituel. Ceci justifie l'introduction d'une covariance normalisée, appelée **coefficient de corrélation**, obtenue en divisant la covariance par la dispersion statistique des valeurs *x* et *y* :

$$correlation\ coefficient(x, y) = \frac{covariance(x, y)}{dispersion(x) * dispersion(y)}$$

Le coefficient de corrélation est toujours compris entre -1 et 1. Une valeur 0 correspond à l'absence de dépendance et une valeur proche de 1 en valeur absolue à une grande dépendance entre les variables *x* et *y*. Lorsque le coefficient est proche de 1, les variables sont dépendantes et les valeurs de *y* augmentent lorsque *x* augmente. Lorsque le coefficient de corrélation est proche de -1, les valeurs de *y* diminuent lorsque *x* augmente.

Reconduisons l'expérience du double jet en étudiant le degré de dépendance entre les deux jets.

```
from random import randint
from gpanel import *
import math

z = 10000 # number of double rolls
k = 10 # scalefactor

def dec2(x):
    return str(round(x, 2))

def mean(xval):
    n = len(xval)
    sum = 0
    for i in range(n):
        sum += xval[i]
    return sum / n

def covariance(xval, yval):
    n = len(xval)
    xm = mean(xval)
    ym = mean(yval)
    cxy = 0
    for i in range(n):
       cxy += (xval[i] - xm) * (yval[i] - ym)
    return cxy / n

def deviation(xval):
    n = len(xval)
    xm = mean(xval)
    sx = 0
    for i in range(n):
```

```
        sx += (xval[i] - xm) * (xval[i] - xm)
    sx = math.sqrt(sx / n)
    return sx

def correlation(xval, yval):
    return covariance(xval, yval) / (deviation(xval) * deviation(yval))

makeGPanel(-1 * k, 11 * k, -2 * k, 14 * k)
title("Double rolls. Independent random variables.")
addStatusBar(30)
drawGrid(0, 10 * k, 0, 13 * k, 10, 13, "gray")

xval =  [0] * z
yval =  [0] * z
xyval = [0] * z

for i in range(z):
    a = k * randint(1, 6)
    b = k * randint(1, 6)
    xval[i] = a
    yval[i] = a + b
    xyval[i] = xval[i] * yval[i]
    move(xval[i], yval[i])
    fillCircle(0.2 * k)

xm = mean(xval)
ym = mean(yval)
xym = mean(xyval)
c = xym - xm * ym
setStatusText("E(x) = " + dec2(xm) + \
          ",  E(y) = " + dec2(ym) + \
          ",  E(x, y) = " + dec2(xym) + \
          ",  covariance = " + dec2(covariance(xval, yval)) + \
          ",  correlation = " + dec2(correlation(xval, yval)))
```

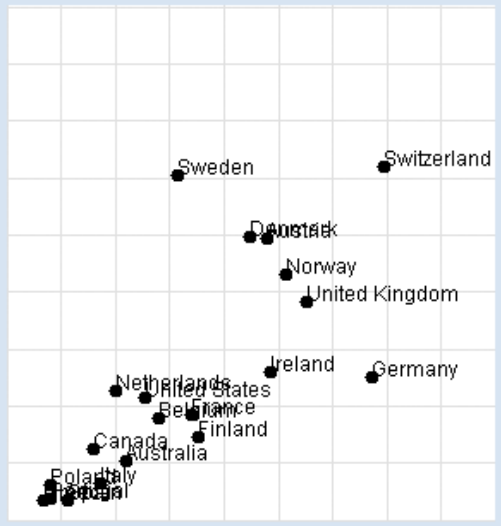
MEMENTO

Même en modifiant le facteur d'échelle *k* de la valeur des faces du dé, le coefficient de corrélation va rester aux alentours de 0.71 au contraire de la covariance qui va beaucoup changer en conséquence Le coefficient de corrélation ne mesure que la dépendance linéaire des variables. De ce fait, il n'est pas capable de mesurer correctement la relation de dépendance entre deux variables si cette dépendance n'est pas linéaire. C'est le cas par exemple dans l'exemple quadratique développé plus haut. Au lieu de parler de coefficient de corrélation, on peut aussi simplement utiliser le terme **corrélation**.

MEDICAL RESEARCH PUBLICATION

Grâce aux connaissances développées dans ce chapitre, vous êtes déjà capables de comprendre et d'évaluer une publication scientifique publiée dans la prestigieuse revue " New England Journal of Medicine ". Celle-ci cherche à savoir s'il y a un lien entre la consommation de chocolat et le nombre de prix Nobel dans différents pays industrialisés. Autrement dit, cette étude cherche à savoir si l'intelligence et la consommation de chocolat sont corrélées d'une quelconque manière. L'étude s'appuie sur les données suivantes librement disponibles sur le Web:

On aimerait reproduire l'étude en utilisant un programme informatique qui utilise une liste *data* contenant des sous-listes de trois éléments : le nom du pays, le nombre de kilogrammes de chocolat consommés annuellement par habitant et le nombre de prix Nobel par tranche de 10 millions d'habitants. Le programme représente les données graphiquement et détermine le coefficient de corrélation entre les deux grandeurs (consommation de chocolat et nombre de prix Nobel.



Prix Nobel :
http://en.wikipedia.org/wiki/List_of_countries_by_Nobel_laureates_per_capita

Consommation de chocolat :
http://www.chocosuisse.ch/web/chocosuisse/en/documentation/facts_figures.html

MEMENTO

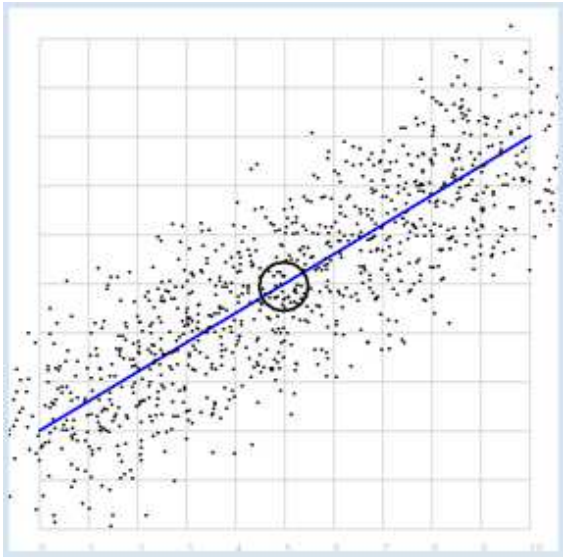
L'étude fournit un coefficient de corrélation d'environ 0.71, ce qui est assez élevé. Il faut cependant prendre du recul pour réaliser que ce résultat peut être interprété de diverses manières. Les deux jeux de données sont manifestement corrélés mais les causes de cette corrélation ne sont pas claires et restent du domaine de la spéculation. Cette étudie ne constitue en aucun cas une preuve que la consommation de chocolat rend plus intelligent. De manière générale, lorsque les grandeurs x et y sont fortement corrélées, il se peut que x soit la cause de y ou l'inverse. Dans notre cas, il se peut que les gens soient plus intelligents parce qu'ils consomment plus de chocolat mais il se peut aussi qu'ils consomment plus de chocolat parce qu'ils sont plus intelligents. Le résultat peut encore être interprété de manière complètement différente si l'on considère que les deux grandeurs sont certainement toutes deux des conséquences d'autres grandeurs non prises en compte dans l'étude.

Prenez le temps de discuter du sens et du but d'une telle étude avec d'autres personnes de votre entourage en vous intéressant à leur opinion sur le sujet.

ERREURS DE MESURE ET BRUIT

Même si la relation entre x et y est parfaitement connue, des erreurs de mesure ou d'autres influences peuvent faire fluctuer les valeurs mesurées. Dans la simulation suivante, on considère une relation linéaire entre x et y où les valeurs y sont sujettes à des fluctuations obéissant à une loi normale.

La simulation détermine le coefficient de corrélation entre x et y et entoure le point de coordonnées (xm, ym) où xm et ym correspondent à l'espérance mathématique de x et y.



```
import random
from gpanel import *
import math

z = 1000
a = 0.6
b = 2
sigma = 1

def f(x):
    y = a * x + b
    return y

def dec2(x):
    return str(round(x, 2))

def mean(xval):
    n = len(xval)
    sum = 0
    for i in range(n):
        sum += xval[i]
    return sum / n

def covariance(xval, yval):
    n = len(xval)
    xm = mean(xval)
    ym = mean(yval)
    cxy = 0
    for i in range(n):
       cxy += (xval[i] - xm) * (yval[i] - ym)
    return cxy / n

def deviation(xval):
    n = len(xval)
    xm = mean(xval)
    sx = 0
    for i in range(n):
        sx += (xval[i] - xm) * (xval[i] - xm)
    sx = math.sqrt(sx / n)
    return sx

def correlation(xval, yval):
    return covariance(xval, yval) / (deviation(xval) * deviation(yval))

makeGPanel(-1, 11, -1, 11)
title("y = 0.6 * x + 2 normally distributed measurement errors")
addStatusBar(30)
drawGrid(0, 10, 0, 10, "gray")
setColor("blue")
lineWidth(3)
line(0, f(0), 10, f(10))

xval = [0] * z
yval = [0] * z

setColor("black")
for i in range(z):
    x = i / 100
    xval[i] = x
    yval[i] = f(x) + random.gauss(0, sigma)
    move(xval[i], yval[i])
    fillCircle(0.03)

xm = mean(xval)
ym = mean(yval)
move(xm, ym)
circle(0.5)
```

```
setStatusText("E(x) = " + dec2(xm) + \
              ",  E(y) = " + dec2(ym) + \
              ",  correlation = " + dec2(correlation(xval, yval)))
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

On voit, comme attendu, que la simulation donne un coefficient de corrélation élevé. Il devient d'ailleurs de plus en plus élevé à mesure que l'on réduit le paramètre de dispersion *sigma*. La corrélation vaut exactement 1 pour *sigma = 0*. De plus, le point ayant pour coordonnées les espérances *P(0.5, 0.5)* est situé sur la droite.

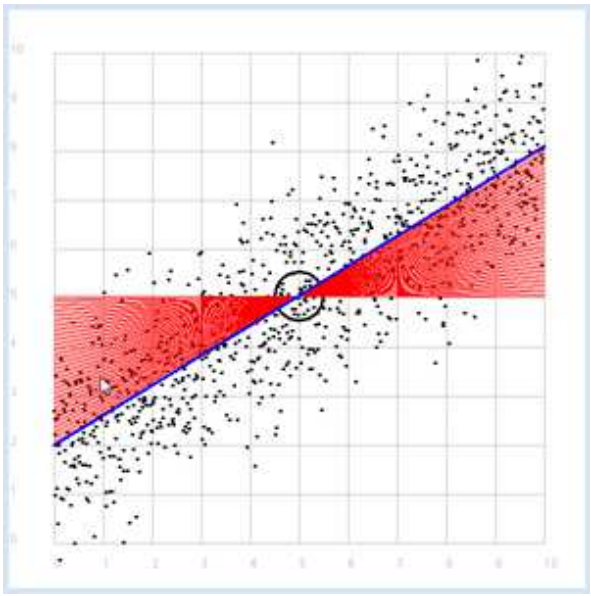
DROITE DE RÉGRESSION, MEILLEUR AJUSTEMENT

Dans l'exemple précédent, on a engendré un nuage de points en ajoutant artificiellement du bruit sur des données correspondant à la base parfaitement à un modèle linéaire. Nous allons maintenant considérer le cas inverse : comment retrouver la droite parfaite qui approxime le mieux un nuage de points résultant de données suivant un modèle linéaire mais entachées d'erreurs ? La droite recherchée est appelée **droite de régression**.

Nous savons déjà au moins une chose importante : la droite de régression passe par le point *P* dont les coordonnées correspondent à l'espérance de *x* et de *y*. Pour la déterminer complètement, on peut procéder de la manière suivante :

On trace une droite quelconque passant par le point *P* et l'on détermine, pour toutes les mesures *x*, les carrés des distances verticales entre les *yi = f(xi)* et la valeur mesurée pour ce même *x*. On détermine la somme de ces erreurs en additionnant tous les carrés de écarts.

On peut ensuite déterminer la meilleure droite de régression dans la simulation en faisant progressivement tourner la droite de test autour du point *P* et en calculant à chaque fois la somme des carrés des erreurs qui va alors diminuer. Dès que cette erreur se remet à augmenter, on sait que la droite précédente était le **meilleur ajustement** recherché.



```
import random
from gpanel import *
import math

z = 1000
a = 0.6
b = 2

def f(x):
    y = a * x + b
    return y

def dec2(x):
    return str(round(x, 2))
```

```
def mean(xval):
    n = len(xval)
    sum = 0
    for i in range(n):
        sum += xval[i]
    return sum / n

def covariance(xval, yval):
    n = len(xval)
    xm = mean(xval)
    ym = mean(yval)
    cxy = 0
    for i in range(n):
       cxy += (xval[i] - xm) * (yval[i] - ym)
    return cxy / n

def deviation(xval):
    n = len(xval)
    xm = mean(xval)
    sx = 0
    for i in range(n):
        sx += (xval[i] - xm) * (xval[i] - xm)
    sx = math.sqrt(sx / n)
    return sx

sigma = 1

makeGPanel(-1, 11, -1, 11)
title("Simulate data points. Press a key...")
addStatusBar(30)
drawGrid(0, 10, 0, 10, "gray")
setStatusText("Press any key")

xval = [0] * z
yval = [0] * z

for i in range(z):
    x = i / 100
    xval[i] = x
    yval[i] = f(x) + random.gauss(0, sigma)
    move(xval[i], yval[i])
    fillCircle(0.03)

getKeyWait()
xm = mean(xval)
ym = mean(yval)
move(xm, ym)
lineWidth(3)
circle(0.5)

def g(x):
    y = m * (x - xm) + ym
    return y

def errorSum():
    sum = 0
    for i in range(z):
        x = i / 100
        sum += (yval[i] - g(x)) * (yval[i] - g(x))
    return sum

m = 0
setColor("red")
lineWidth(1)
error_min = 0
while m < 5:
    line(0, g(0), 10, g(10))
    if m == 0:
```

```
        error_min = errorSum()
    else:
        if errorSum() < error_min:
            error_min = errorSum()
        else:
            break
    m += 0.01

title("Regression line found")
setColor("blue")
lineWidth(3)
line(0, g(0), 10, g(10))
setStatusText("Found slope: " +  dec2(m) + \
               ", Theory: " + dec2(covariance(xval, yval)
              /(deviation(xval) * deviation(xval))))
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Au lieu d'utiliser une simulation informatique pour déterminer la droite de régression qui donne le meilleur ajustement, on peut également calculer sa pente grâce à la formule

$$m = \frac{covariance(x, y)}{dispersion(x)^2}$$

Puisque la droite de régression passe par le point *P* de coordonnées *E(x)* et *E(y)*, elle est donnée par l'équation cartésienne :

$$y - E(y) = m * (x - E(x))$$

EXERCICES

1. La célèbre loi d'Engel issue des sciences économiques stipule qu'en moyenne, lorsque le revenu des ménages augmente, le montent mensuel absolu dépensé pour la nourriture augmente mais que ce montant diminue relativement aux autres dépenses.
   a. Visualiser la relation entre le revenu et les dépenses totales pour la nourriture
   b. Faire de même pour la relation entre le revenu et les dépenses relatives pour la nourriture
   c. Déterminer le coefficient de corrélation entre le revenu et les dépenses absolues pour la nourriture
   d. Déterminer la droite de régression entre le revenu et les dépenses absolues pour la nourriture en utilisant les données ci-dessous

   Data:

Revenu mensuel	4000	4100	4200	4300	4400	4500	4600	4700	4800	4900
Dépenses%	64	63.25	62.55	61.90	61.30	60.75	60.25	59.79	59.37	58.99

5000	5100	5200	5300	5400	5500	5600	5700	5800	5900	6000
58.65	58.35	58.08	57.84	57.63	57.45	57.30	57.17	57.06	56.97	56.90

2. L'histoire évolutionniste de l'univers acceptée par la communauté scientifique présuppose que tout a commencé il y a très longtemps par un Big Bang et que, depuis lors, l'univers est en expansion. La question principale est de savoir à quand remonte le Big Bang ou, autrement dit, quel est l'âge de l'univers. L'astronome Hubble a publié une fameuse étude en 1929 dans laquelle il montre qu'il y a une relation linéaire entre la distance *d* séparant les galaxies et leur vitesse d'éloignement *v*. La loi de Hubble est formulée de la manière suivante :

v = H * d

où H est la constante de Hubble.

On peut comprendre le processus de pensée des astrophysiciens en partant de données suivantes produites par le télescope spatial Hubble.

Galaxie	Distance [Mpc)	Vetesse [km/s]
NGC0300	2	133
NGC095	9.16	664
NGC1326A	16.14	1794
NGC1365	17.95	1594
NGC1425	21.88	1473
NGC2403	3.22	278
NGC2541	11.22	714
NGC2090	11.75	882
NGC3031	3.63	80
NGC3198	13.8	772
NGC3351	10	642
NGC3368	10.52	768
NGC3621	6.64	609
NGC4321	15.21	1433
NGC4414	17.7	619
NGC4496A	14.86	1424
NGC4548	16.22	1384
NGC4535	15.78	1444
NGC4536	14.93	1423
NGC4639	21.98	1403
NGC4725	12.36	1103
IC4182	4.49	318
NGC5253	3.15	232
NGC7331	14.72	999

1 Megaparsec (Mpc) = 3.09 * 10^{19}km

a) Représenter ces données dans un nuage de points et les copier dans la liste *data* au sein du programme.

```
data = [
["NGC0300", 2.00, 133],
["NGC095", 9.16, 664],
["NGC1326A", 16.14, 1794],
["NGC1365", 17.95, 1594],
["NGC1425", 21.88, 1473],
["NGC2403", 3.22, 278],
["NGC2541", 11.22, 714],
["NGC2090", 11.75, 882],
["NGC3031", 3.63, 80],
["NGC3198", 13.80, 772],
["NGC3351", 10.0, 642],
["NGC3368", 10.52, 768],
["NGC3621", 6.64, 609],
["NGC4321", 15.21, 1433],
["NGC4414", 17.70, 619],
["NGC4496A", 14.86, 1424],
["NGC4548", 16.22, 1384],
["NGC4535", 15.78, 1444],
```

["NGC4536", 14.93, 1423],
["NGC4639", 21.98, 1403],
["NGC4725", 12.36, 1103],
["IC4182", 4.49, 318],
["NGC5253", 3.15, 232],
["NGC7331", 14.72, 999]]

Données tirées de Freedman et al, The Astrophysical Journal, 553 (2001)

b) Montrer que les données sont bien corrélées et déterminer la pente H de la droite de régression.

c) En supposant que la vitesse *v* d'une certaine galaxie demeure constante, sa distance est alors donnée par *d* = *v* * *T*, où *T* est l'âge de l'univers. On peut donc utiliser la loi de Hubble (v = H*d) pour estimer l'âge de l'univers puisque T = 1 / H. Déterminer l'âge de l'univers T donné par ce modèle linéaire.

.

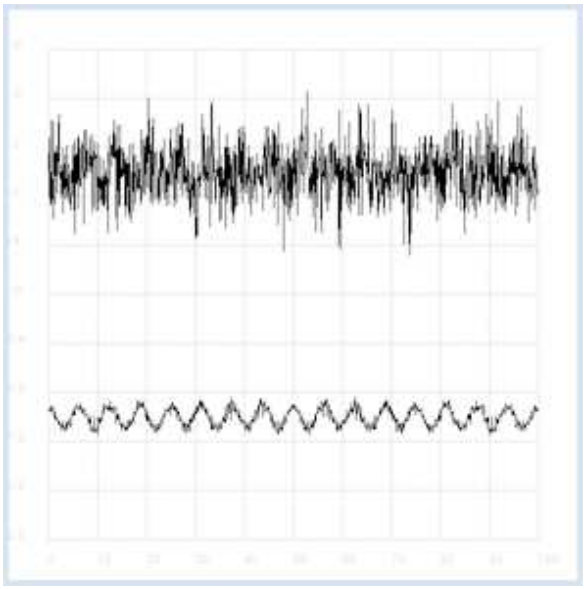
MATÉRIEL SUPPLÉMENTAIRE

DÉTECTER DES INFORMATIONS CACHÉES EN UTILISANT L'AUTOCORRÉLATION

Des êtres intelligents habitant sur une autre planète désirent contacter d'autres êtres vivants en émettant des signaux radio présentant une certaine régularité (vous pouvez vous imaginez une sorte de code morse). L'intensité du signal faiblit au fur et à mesure de son voyage aux confins de l'univers s'en retrouve masqué par un bruit présentant des fluctuations statistiques. Le signal est capté sur terre à l'aide d'un radiotélescope qui ne révèle plus que du bruit.

Les statistiques et un programme informatique peuvent nous permettre de reconstituer le signal original. Si l'on calcule le coefficient de corrélation du signal avec lui-même mais décalé dans le temps, on peut réduire la part du bruit dans le signal. La corrélation d'une grandeur avec elle-même est appelée **autocorrélation**. On peut simuler ces propriétés importantes pour le traitement de signal à l'aide d'un programme Python.

Le signal utile original est une fonction sinusoïdale sur laquelle on superpose du bruit en ajoutant à chaque valeur de l'échantillon un nombre aléatoire selon une loi normale. Le programme ci-dessous commence par représenter le signal comportant le bruit aléatoire dans la partie supérieure de l'écran et attend que l'utilisateur presse une touche du clavier. On peut constater que le signal n'est plus reconnaissable.



Dans un deuxième temps, le programme construit l'autocorrélation du signal avec lui-même et représente l'évolution du coefficient de corrélation dans la partie inférieure du graphe. Le signal utile original sera à nouveau clairement reconnaissable.

```
import random
from gpanel import *
import math

def mean(xval):
    n = len(xval)
    sum = 0
    for i in range(n):
        sum += xval[i]
    return sum / n

def covariance(xval, yval):
    n = len(xval)
    xm = mean(xval)
    ym = mean(yval)
    cxy = 0
    for i in range(n):
       cxy += (xval[i] - xm) * (yval[i] - ym)
    return cxy / n

def deviation(xval):
    n = len(xval)
    xm = mean(xval)
    sx = 0
    for i in range(n):
        sx += (xval[i] - xm) * (xval[i] - xm)
    sx = math.sqrt(sx / n)
    return sx

def correlation(xval, yval):
    return covariance(xval, yval)/(deviation(xval)*deviation(yval))
def shift(offset):
    signal1 = [0] * 1000
    for i in range(1000):
       signal1[i] = signal[(i + offset) % 1000]
    return signal1

makeGPanel(-10, 110, -2.4, 2.4)
title("Noisy signal. Press a key...")
drawGrid(0, 100, -2, 2.0, "lightgray")

t = 0
dt = 0.1
signal = [0] * 1000
while t < 100:
    y = 0.1 * math.sin(t) # Pure signal
#    noise = 0
    noise = random.gauss(0, 0.2)
    z = y + noise
    if t == 0:
        move(t, z + 1)
    else:
        draw(t, z + 1)
    signal[int(10 * t)] = z
    t += dt

getKeyWait()
title("Signal after autocorrelation")
for di in range(1, 1000):
    y = correlation(signal, shift(di))
    if di == 1:
        move(di / 10, y - 1)
    else:
        draw(di / 10, y - 1)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

Pour rendre cela encore plus tangible, commencez par écouter un rendu sonore du signal bruité

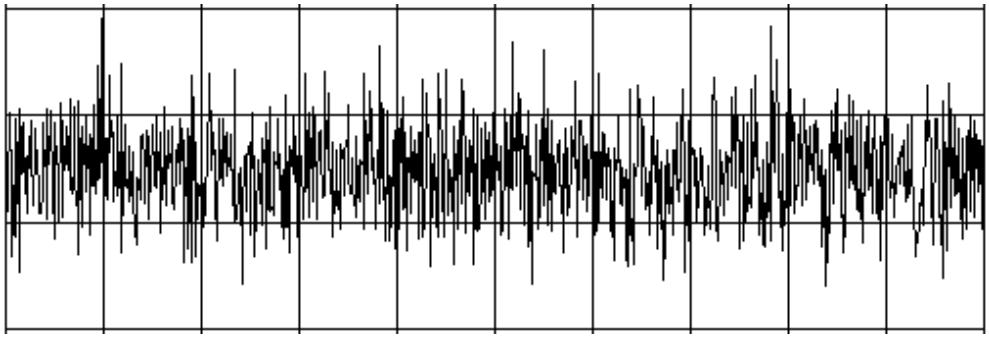
puis du signal reconstitué par l'autocorrélation. Pour ce faire, il faudra vous replonger dans le chapitre traitant du on **son**.

```
from soundsystem import *
import math
import random
from gpanel import *

n = 5000

def mean(xval):
    sum = 0
    for i in range(n):
        sum += xval[i]
    return sum / n

def covariance(xval, k):
    cxy = 0
    for i in range(n):
       cxy += (xval[i] - xm) * (xval[(i + k) % n] - xm)
    return cxy / n

def deviation(xval):
    xm = mean(xval)
    sx = 0
    for i in range(n):
        sx += (xval[i] - xm) * (xval[i] - xm)
    sx = math.sqrt(sx / n)
    return sx

makeGPanel(-100, 1100, -11000, 11000)
drawGrid(0, 1000, -10000, 10000)
title("Press <SPACE> to repeat. Any other key to continue.")

signal = []
for i in range(5000):
    value = int(200 *  (math.sin(6.28 / 20 * i) + random.gauss(0, 4)))
    signal.append(value)
    if i == 0:
        move(i, value + 5000)
    elif i <= 1000:
        draw(i, value + 5000)

ch = 32
while ch == 32:
    openMonoPlayer(signal, 5000)
    play()
    ch = getKeyCodeWait()

title("Autocorrelation running. Please wait...")
signal1 = []
xm = mean(signal)
sigma = deviation(signal)
q = 20000 / (sigma * sigma)
for di in range(1, 5000):
    value = int(q *  covariance(signal, di))
    signal1.append(value)
title("Autocorrelation Done. Press any key to repeat.")
for i in range(1, 1000):
    if i == 1:
        move(i, signal1[i] - 5000)
    else:
        draw(i, signal1[i] - 5000)

while True:
    openMonoPlayer(signal1, 5000)
    play()
    getKeyCodeWait()
```

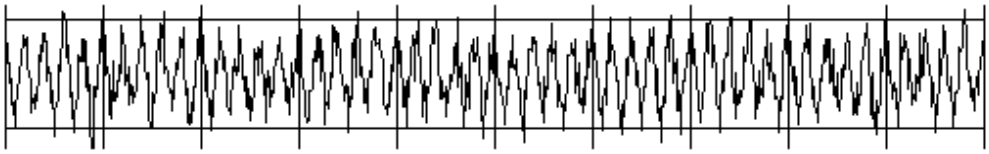
MEMENTO

Au lieu d'exécuter le programme, vous pouvez également écouter les deux signaux sonores en les téléchargeant au format WAV :

Signal bruité **(cliquer ici**)



Signal désiré **(cliquer ici**)



8.7 NOMBRES COMPLEXES & FRACTALES

INTRODUCTION

Les nombres complexes sont très importants en mathématiques du fait qu'ils constituent une extension de l'ensemble des nombres réels et permettent ainsi de formuler de nombreuses propositions de manière plus aisée et plus générale. Ils jouent également un rôle très important en sciences et dans les domaines technologiques, particulièrement en physique et en génie électrique [**plus...**]. Fort heureusement, les nombres complexes sont intégrés de base dans l'interpréteur Python qui contient des opérateurs permettant d'effectuer l'addition, la soustraction, la multiplication et la division de nombres complexes. De plus, le module *cmath* comporte de nombreuses fonctions acceptant des arguments complexes.

Les nombres complexes peuvent être représentés dans le plan complexe par de points ou des vecteurs. TigerJython permet d'utiliser une fenêtretortue, un canevas *GPanel* ou une grille de jeu *JGameGrid* pour représenter le plan complexe car toutes les fonctions de ces trois bibliothèques respectives demandant des paramètres de coordonnées (x, y) acceptent également des nombres complexes à la place de deux coordonnées réelles.

CONCEPTS DE PROGRAMMATION: *Type de données de nombres complexes, application conforme, fractale de Mandelbrot*

OPÉRATIONS ÉLÉMENTAIRES SUR LES NOMBRES COMPLEXES

En Python, l'unité imaginaire est dénotée par le symbole *j* au lieu de *i* car c'est ainsi qu'il est noté la plupart du temps en génie électrique. Il y a plusieurs manières de définir un nombre complexe dont la partie réelle vaut 2 et la partie imaginaire vaut 3 :

```
z = 2 + 3j          or      z = 2 + 3 * 1j          or      z  = complex(2, 3)
```

Pour tester les exemples suivants et vous faire la main avec les nombres complexes, le mieux est d'utiliser la console *TigerJython* :

On peut obtenir la partie réelle et la partie imaginaire d'un nombre complexe grâce à *z.real* et *z.imag* respectivement. Il faut garder à l'esprit qu'il ne s'agit pas d'appels de méthodes mais que *z.real* et *z.imag* se comportent comme des variables de type *float* en lecture seule.

```
>>> z = 2 + 3j
>>> z
    (2+3j)
>>> z.real
    2.0
>>> z.imag
    3.0
```

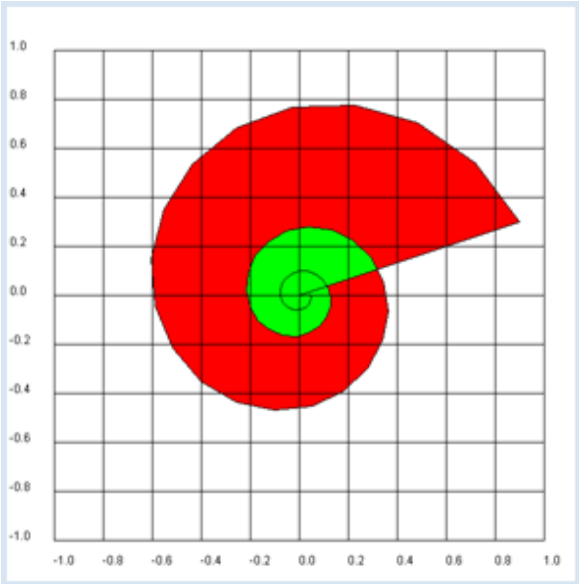
Le carré de l'unité imaginaire *1j* est -1. En d'autres termes, l'unité imaginaire correspond à la racine de -1. Pour calculer la racine carrée d'un nombre complexe, il faut importer le module *cmath* au lieu du module *math*.

```
>>> z = 1j
>>> z * z
    (-1+0j)
>>> import cmath
>>> cmath.sqrt(-1)
    1j
```

La fonction standard *abs()* retourne non seulement la valeur absolue d'un nombre entier ou flottant, mais également le module d'un nombre complexe. Les opérations élémentaires +, -, *, / et l'opérateur d'exponentiation ** sont également valables pour les nombres complexes. Ces opérations présentent les mêmes règles de précédence que dans le cas des nombres à virgule flottante.

```
>>> z = 3 + 4j
>>> abs(z)
    5.0
>>> 2 * (z + 1) - z
    (5+4j)
>>> z**2 / z
    (3+4j)
```

Le programme suivant représente graphiquement les puissances du nombre complexe z = 0.9 + 0.3j dont le module est légèrement inférieur à 1. Puisque le module du produit de deux nombres complexes est donné par le produit de leur module et que l'argument (aussi appelé la phase) du produit est la somme des phases, les puissances de *z* vont former une spirale facilement représentable à l'aide de *GPanel*. Pensez à faire le remplissage de la spirale avant de dessiner le quadrillage car sinon, le quadrillage viendrait perturber le remplissage puisqu'il contribue à déterminer la zone fermée à colorier.




```
from gpanel import *
makeGPanel(-1.2, 1.2, -1.2, 1.2)
title("Complex plane")

z = 0.9 + 0.3j
for n in range(1, 60):
    y = z**n
    draw(y)
fill(0.2, 0, "white", "red")
fill(0.0, 0.2, "white", "green")
drawGrid(-1.0, 1.0, -1.0, 1.0)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

L'appel *draw(z)* a le même effet que *draw(z.real, z.imag)* mais en plus simple. Afin de pouvoir dessiner les axes, on spécifie, lors de la création du *GPanel,* une plage de coordonnées 10% supérieure à la plage de coordonnées nécessaires pour tracer le graphe. Pour que les unités des axes soient des nombres à virgule, il faut spécifier les coordonnées en nombre flottant dans l'appel à *drawGrid()*.

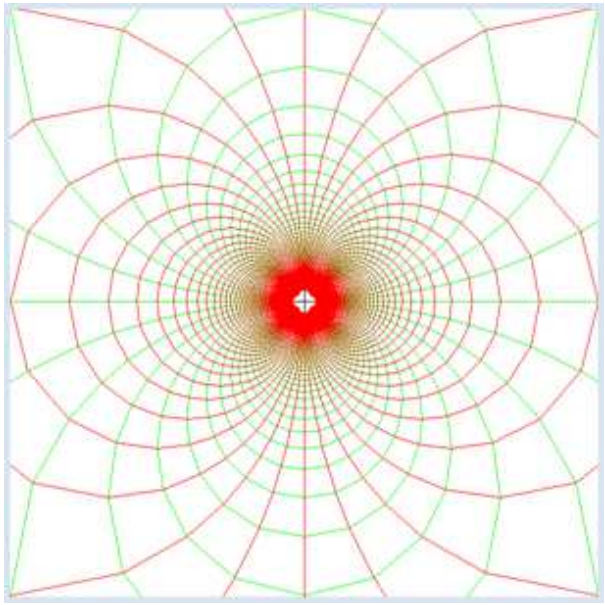
APPLICATIONS CONFORMES

Considérons maintenant des applications qui, à tout point *P(x, y)* du plan (pixel), associent un et un seul point (pixel) *P'(x', y')*.

On peut également considérer les points du plan comme des nombres complexes et considérer les applications du plan dans lui-même comme des fonctions complexes qui, à tout nombre complexe *z*, associent un et un seul nombre complexe *z'*. On écrit donc

z' = f(z)

Dans le programme suivant, on représente la fonction complexe *z' = f(z) = 1/z* (inversion) que l'on évalue sur l'ensemble des points formant une grille du plan complexe.



On choisit donc dans le plan complexe une plage entre -5 et 5 sur chacun des axes et on imagine un grillage constitué de 201 lignes horizontales et verticales distantes de 1/20 entre -5 et 5. On dessine en vert l'image par la fonction *f* des lignes horizontales et en rouge l'image des lignes verticales. On obtient ainsi une magnifique image permettant de mieux se représenter l'effet de la fonction *f*.

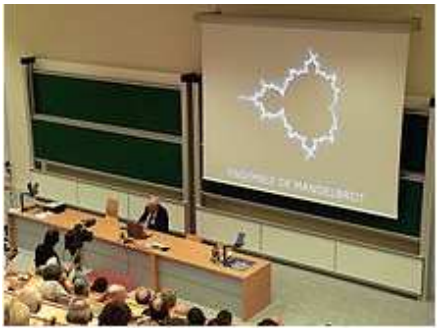
```
from gpanel import *

# function f(z) = 1/z
def f(z):
    if z == 0:
        return 0
    return 1 / z

min = -5.0
max = 5.0
step = 1 / 20
reStep = complex(step, 0)
imStep = complex(0, step)

makeGPanel(min, max, min, max)
title("Conformal mapping for f(z) = 1 / z")
line(min, 0, max, 0)  # Real axis
line(0, min, 0, max) # Imaginary axis

# Transform horizontal line per line
setColor("green")
z = complex(min, min)
while z.imag < max:
    z = complex(min, z.imag) # left
    move(f(z))
    while z.real < max: # move along horz. line
        draw(f(z))
        z = z + reStep
    z = z + imStep

# Transform vertical line per line
setColor("red")
z = complex(min, min)
while z.real < max:
    z = complex(z.real, min) # bottom
    move(f(z))
    while z.imag < max:  # move along vert. line
        draw(f(z))
        z = z + imStep
    z = z + reStep
```

MEMENTO

Ce qu'il faut absolument retenir de cette figure est que les courbes obtenues en appliquant la fonction *f* à des lignes perpendiculaires se coupent à angle droit. Une application qui conserve les angles est appelée **conforme** [**plus...**] .

MANDELBROT FRACTALS

Nombreuses sont les personnes connaissant les fractales et vous avez sans doute déjà entendu parler de **l'ensemble de Mandelbrot**. Nous allons le représenter à l'aide d'un programme. De nombreux algorithmes permettant de représenter les fractales sont basés sur les nombres complexes, ce qui nous motive à travailler avec les nombres complexes en Python. Le père de la géométrie des fractales est Benoît Mandelbrot (1924-2010).



Mandelbrot lors du cours d'introduction de la Légion d'honneur (2006) (© Wiki)

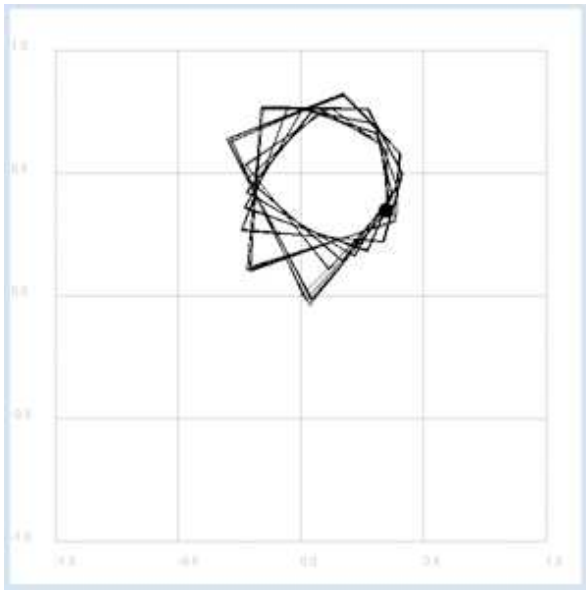
Pour générer une fractale de Mandelbrot, on considère une suite de nombres complexes définie par un nombre complexe quelconque *c* et la relation de récurrence

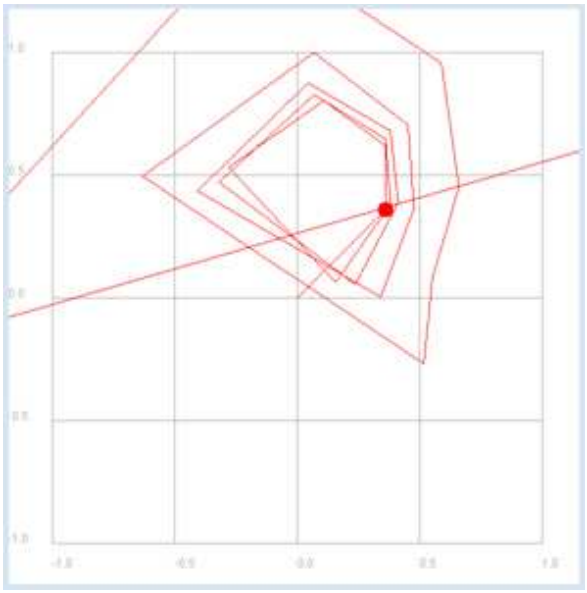
$z' = z^2 + c$ avec le terme initial $z_0 = 0$

L'ensemble de Mandelbrot est formé de tous les nombres complexes *c* tels que la suite précédente est bornée.

Avant d'attaquer l'algorithme permettant de représenter l'ensemble de Mandelbrot, représentons les termes des suites définies par les nombres c1 = 0.35 + 0.35j et c2 = 0.36 + 0.36j.

On peut immédiatement constater que *c1* appartient à l'ensemble de Mandelbrot et que *c2* n'en fait pas partie.





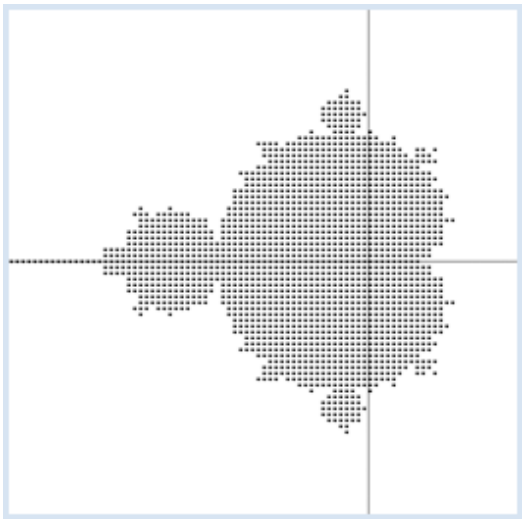
```
from gpanel import *

def f(z):
    return z * z + c

makeGPanel(-1.2, 1.2, -1.2, 1.2)
```

```
title("Mandelbrot iteration")

drawGrid(-1, 1.0, -1, 1.0, 4, 4, "gray")

isMandelbrot = askYesNo("c in Mandelbrot set?")
if isMandelbrot:
    c = 0.35 + 0.35j
    setColor("black")
else:
    c = 0.36+ 0.36j
    setColor("red")

title("Mandelbrot iteration with c = " + str(c))
move(c)
fillCircle(0.03)

z = 0j
while True:
    if z == 0:
        move(z)
    else:
        draw(z)
    z = f(z)
    delay(100)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

Pour déterminer quels points d'une certaine région du plan complexe appartiennent à l'ensemble de Mandelbrot, il suffit d'effectuer le test précédent pour tous les points appartenant à ce domaine. Comme cela n'est pas possible de manière exhaustive, on se retreint à un nombre fini et raisonnable de points.

Dans le programme suivant, ont fait l'hypothèse quelque peu réductrice qu'un nombre *c* fait partie de l'ensemble de Mandelbrot si le module des 50 premiers termes de la suite qu'il définit sont tous inférieurs à 2.

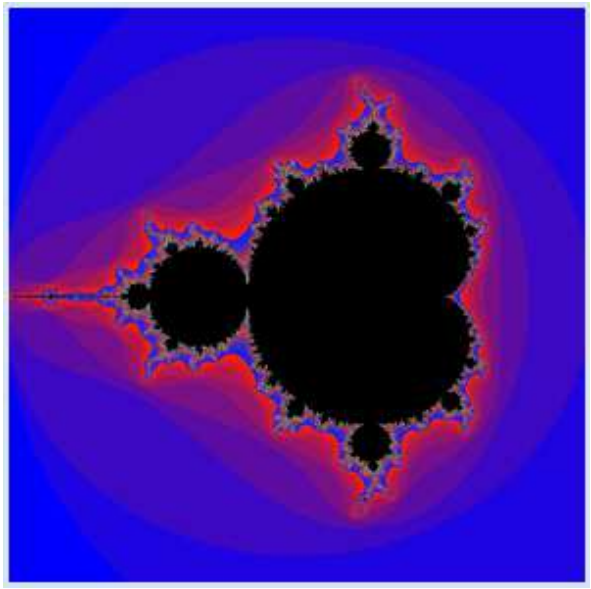


```
from gpanel import *

def isInSet(c):
    z = 0
    for n in range(maxIterations):
        z = z*z + c
        if abs(z) > R: # diverging
            return False
    return True

maxIterations = 50
R = 2
xmin = -2
xmax = 1
xstep = 0.03
ymin = -1.5
ymax = 1.5
ystep = 0.03

makeGPanel(xmin, xmax, ymin, ymax)
line(xmin, 0, xmax, 0)  # real axis
```

```
line(0, ymin, 0, ymax) # imaginary axis
title("Mandelbrot set")
y = ymin
while y <= ymax:
    x = xmin
    while x <= xmax:
        c = x + y*1j
        inSet = isInSet(c)
        if inSet:
            move(c)
            fillCircle(0.01)
        x += xstep
    y += ystep
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

Ce graphique permet déjà de déceler l'ensemble de Mandelbrot. On peut cependant encore obtenir de plus jolis graphiques si l'on représente en couleur les points *c* pour lesquels la suite des modules diverge. On choisira la couleur en fonction de la vitesse avec laquelle la suite définie par le point *c* diverge. On peut obtenir une bonne indication de cette vitesse de divergence en considérant le rang *itCount* du premier terme de la suite dont le module excède 2.



Pour associer *itCount* à une couleur, on utilise la fonction *getIterationColor()* que vous pouvez bidouiller à votre convenance pour obtenir une magnifique image de fractale.

```
from gpanel import *

def getIterationColor(it):
    color = makeColor((30 * it) % 256,
                      (4 * it) % 256,
                      (255 - (30 * it)) % 256)
    return color

def mandelbrot(c):
    z = 0
    for it in range(maxIterations):
        z = z*z + c
        if abs(z) > R: # diverging
            return it
    return maxIterations

maxIterations = 50
R = 2
xmin = -2
xmax = 1
xstep = 0.003
ymin = -1.5
ymax = 1.5
ystep = 0.003

makeGPanel(xmin, xmax, ymin, ymax)
title("Mandelbrot set")
enableRepaint(False)
y = ymin
while y <= ymax:
    x = xmin
```

```
    while x <= xmax:
        c = x + y*1j
        itCount = mandelbrot(c)
        if itCount == maxIterations: # inside Mandelbrot set
            setColor("black")
        else: # outside Mandelbrot set
           setColor(getIterationColor(itCount))
        point(c)
        x += xstep
    y += ystep
    repaint()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Pour accélérer le dessin, on effectue l'appel *enableRepaint(False)* et on effectue la mise à jour du rendu uniquement à la fin de chaque ligne avec *repaint()*.

Les fractales de Mandelbrot présentent la particularité de répéter à l'infini la même structure lorsque l'on effectue un agrandissement sur une de ses parties [**plus...**] .

EXERCICES

1. On peut obtenir une magnifique fractale si l'on colorie les points du domaine complexe entre -20 et 20 dont le carré du module arrondi à l'unité est pair, c'est-à-dire les points qui vérifient la condition *int(abs(z) * abs(z)) % 2 == 0*. Prendre un incrément de 0.1.

2. Étudier les fonctions complexes suivantes de la même manière que la fonction inverse :

   a) $z' = f(z) = z^2$

   b) $z' = f(z) = a * z$ avec un nombre complexe $a = 2 + 1j$

   c) $z' = f(z) = e^z$

   d) $z' = f(z) = \frac{1 - z}{1 + z}$ (Transformation de Möbius)

   Procéder en représentant l'image par chacune de ces fonctions du grillage formé par les lignes horizontales et verticales espacées de 1/10 entre -5 et 5. Décrire en mots l'image du grillage ainsi obtenue et déterminer si l'application en question est conforme ou non.

3. Représenter quelques ensembles de Mandelbrot en utilisant diverses associations de couleurs, par exemple :

Nombre d'itérations	**Couleur**
< 3	dark gray
< 5	green
< 8	red
< 10	blue
< 100	yellow
sonst	black

MATÉRIEL SUPPLÉMENTAIRE

COURANT ALTERNATIF ET IMPÉDANCE

Les circuits électriques constitués de composants passifs (résistances, condensateurs, self) et traversés par des courants alternatifs peuvent être modélisés comme des circuits à courant continu si l'on utilise des valeurs complexes pour les tensions, les courants et les résistances. Une résistance complexe généralisée est appelée **impédance** et souvent désignée par Z ou par X lorsqu'il s'agit d'une impédance purement imaginaire (partie réelle nulle). L'impédance d'une résistance ohmique est R, celle d'une self $X_L = j\omega L$ (inductance = self) et celle d'un condensateur est donnée par $X_C = 1 / j\omega C$ (C: capacité), où $\omega = 2\pi f$ (f: fréquence).

Une tension alternative complexe u = u(t) parcourt un cercle du plan complexe de manière uniforme (vitesse angulaire constante). Si l'on applique cette tension aux bornes d'une impédance , le courant i(t) la traversant sera déterminé par la loi d'Ohm u = Z * i. Puisque lors de la multiplication deux nombres complexes, les arguments (phases) de deux nombres sont additionnés et les modules multipliés, le courant suit la tension avec un décalage de phase de Z:

phase(u) = phase(Z) + phase(i)

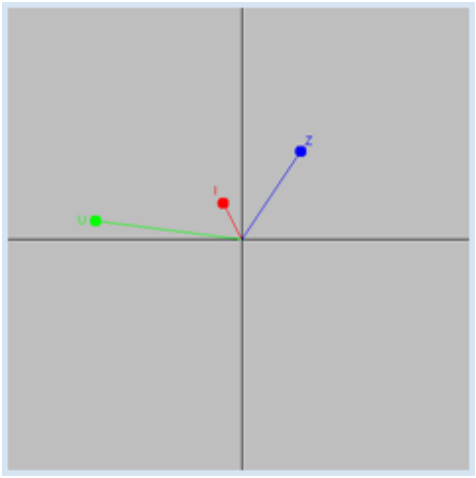
De ce fait, le courant *i* parcourt également un cercle du plan complexe. Les modules (amplitudes) sont donnés par :

$| u | = | Z | * | i |$

Le programme suivant représente cette relation dans le plan complexe en prenant les valeurs

$| u | = 5V$ et $Z = 2 + 3j$

ainsi qu'une fréquence f = 10 Hz. Puisque le graphique serait complètement effacé, reconstitué et redessiné à chaque étape de l'animation en utilisant la fonction *repaint()*, on utilise plutôt un *GPanel* avec *enableRepaint(False)*.




```
from gpanel import *
import math

def drawAxis():
   line(min, 0, max, 0)  # real axis
   line(0, min, 0, max) # imaginary axis


def cdraw(z, color, label):
    oldColor = setColor(color)
    line(0j, z)
    fillCircle(0.2)
    z1 = z + 0.5 * z / abs(z) - (0.1 + 0.2j)
    text(z1, label)
    setColor(oldColor)

min = -10
max = 10
dt = 0.001

makeGPanel(min, max, min, max)
enableRepaint(False)
bgColor("gray")
```

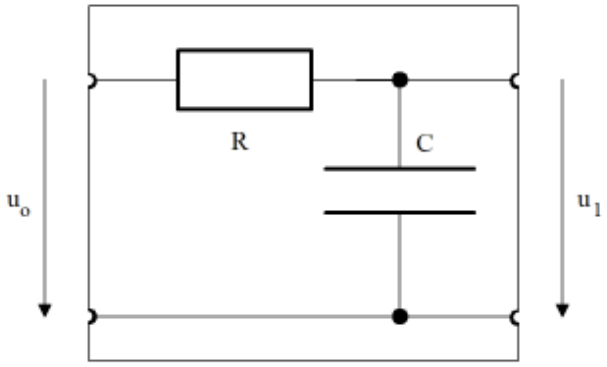
```
title("Complex voltages and currents")

f = 10 # Frequency
omega = 2 * math.pi * f

t = 0
uA = 5
Z = 2 + 3j

while True:
    u = uA * (math.cos(omega * t) + 1j * math.sin(omega * t))
    i = u / Z
    clear()
    drawAxis()
    cdraw(u, "green", "U")
    cdraw(i, "red", "I")
    cdraw(Z, "blue", "Z")
    repaint()
    t += dt
    delay(100)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Les circuits électriques constitués de composants passifs peuvent être traités avec les lois du courant continu si l'on considère les tensions, les courants et les résistances comme des valeurs complexes.

Cette connaissance est applicable à ce simple circuit constitué uniquement d'une résistance et d'un condensateur. On veut connaitre la tension *u1* à sa sortie en fonction de la fréquence *f* pour des valeurs connues de la tension uo en entrée, de la résistance R et de la capacité C.

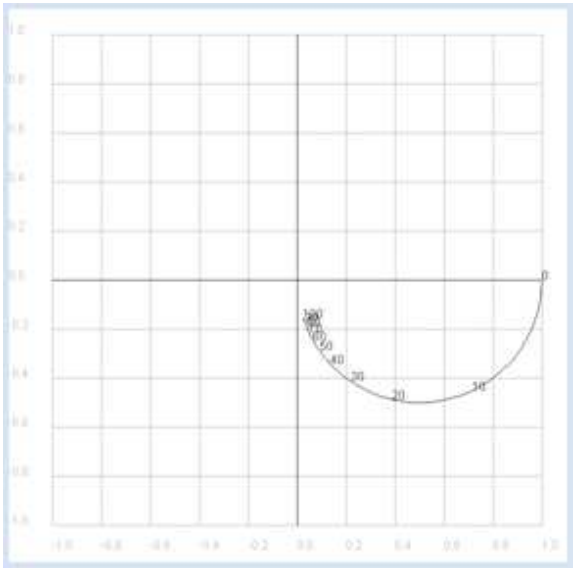


Le calcul est simple : la mise en série de la résistance R et du condensateur C donne lieu à l'impédance Z = R + $X_C$et, de ce fait, au courant i = uo / Z. On obtient la tension de sortie en utilisant la loi d'Ohm généralisée :

$$u_1 = X_c * i = \frac{X_c}{R + X_c} * u_0 = \quad \text{our} \quad u_1 = v * u_0 \quad \text{avec} \quad v = \frac{X_c}{R + X_c}$$

La grandeur complexe *v* est appelée **gain** du circuit électrique. On peut la visualiser dans le plan complexe pour différentes valeurs de *f* et l'on observe alors que le gain vaut 1 pour une fréquence nulle (courant continu) et que ce gain va diminuer jusqu'à attendre 0 lorsqu'on augmente la fréquence de la tension.

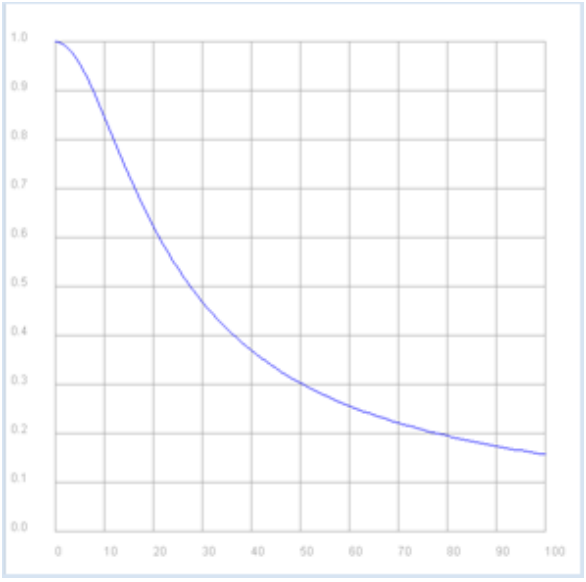
De ce fait, un tel circuit transmet bien les basses fréquences mais atténue les hautes fréquences. On parle de **filtre passe-bas**.

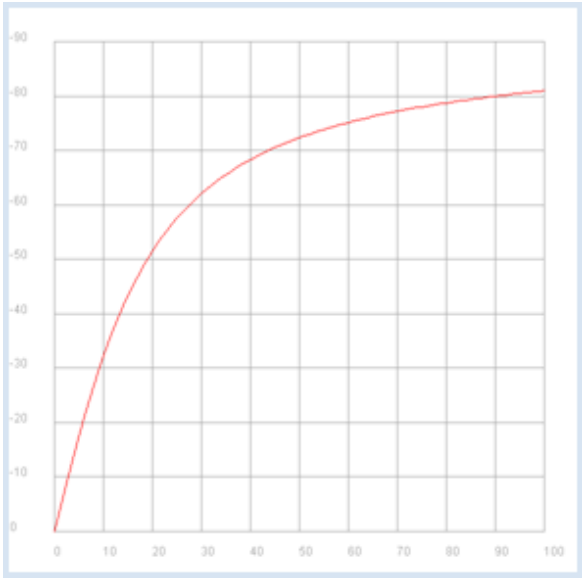


```
from gpanel import *
from math import pi

def drawAxis():
   line(-1, 0, 1, 0)  # Real axis
   line(0, -1, 0, 1) # Imaginary axis

makeGPanel(-1.2, 1.2, -1.2, 1.2)
drawGrid(-1.0, 1.0, -1.0, 1.0, "gray")
setColor("black")
drawAxis()
title("Complex gain factor - low pass")

R = 10
C = 0.001
def v(f):
    if f == 0:
        return 1 + 0j
    omega = 2 * pi * f
    XC = 1 / (1j * omega * C)
    return XC / (R + XC)

f = 0 # Frequency
while f <= 100:
    if f == 0:
        move(v(f))
    else:
        draw(v(f))
    if f % 10 == 0:
        text(str(f))
    f += 1
    delay(10)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

Un diagramme de Bode est constitué de deux graphiques. Dans le premier, on représente l'amplitude du gain en fonction de la fréquence et dans le second, on représente sa phase en fonction de la fréquence. On utilise généralement des échelles logarithmiques.





```
from gpanel import *
import math
import cmath

R = 10
C = 0.001

def v(f):
    if f == 0:
        return 1 + 0j
    omega = 2 * math.pi * f
    XC = 1 / (1j * omega * C)
    return XC / (R + XC)

p1 = GPanel(-10, 110, -0.1, 1.1)
drawPanelGrid(p1, 0, 100, 0, 1.0, "gray")
p1.title("Bode Plot - Low Pass, Gain")
p1.setColor("blue")
f = 0
while f <= 100:
    if f == 0:
        p1.move(f, abs(v(f)))
    else:
        p1.draw(f, abs(v(f)))
    f += 1

p2 = GPanel(-10, 110, 9, -99)
drawPanelGrid(p2, 0, 100, 0, -90, 10, 9, "gray")
p2.title("Bode Plot - Low Pass, Phase")
p2.setColor("red")
f = 0
while f <= 100:
    if f == 0:
        p2.move(f, math.degrees(cmath.phase(v(f))))
    else:
        p2.draw(f, math.degrees(cmath.phase(v(f))))
    f += 1
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Le diagramme de Bode montre encore une fois que le circuit étudié transmet bien les basses fréquences et filtre les hautes fréquences. De plus, il y a un décalage de phase entre le signal d'entrée et le signal de sortie dans la plage entre 0 et -90 degrés.

EXERCICES

1. La fréquence

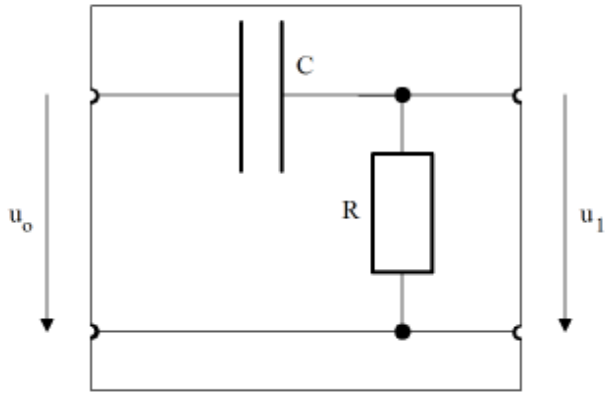
$$f_c = \frac{1}{2 \pi R C}$$

est appelée fréquence de coupure. Montrer que, pour le filtre passe-bas RC étudié précédemment, pour R = 10 Ohm et C = 0.001 F, le gain du circuit pour cette fréquence de coupure vaut :
$1/\sqrt{2}$

2. Le gain est souvent donné en décibels (dB) en prenant le logarithme en base 10 de *v* multiplié par 20 : dB = 20 log |v|. Dessiner le diagramme de Bode pour le filtre passe-bas RC en prenant R = 10 Ohm et C = 0.001 F en utilisant une échelle en dB allant jusqu'à -100 dB et une échelle logarithmique pour la fréquence entre 1Hz et 100 kHz.

   Utiliser le diagramme de Bode pour confirmer que l'atténuation des hautes fréquences atteint 20 dB par décade.

3. La figure ci-dessous représente un **filtre passe-haut** (R = 10 Ohm, C = 0.001 F).



   Comme cela a été fait dans l'exercice 2, écrire un programme qui dessine le diagramme de Bode pour le gain et discuter le comportement au niveau des fréquences.

8.8 ANALYSE SPECTRALE

INTRODUCTION

Dès qu'un rayon lumineux parvient à l'œil ou qu'un son atteint l'oreille, il en découle un signal qui peut être interprété comme une fonction du temps *y(t)*. Lorsqu'il s'agit d'une lumière monochromatique ou d'un son pur, cette fonction sera une oscillation sinusoïdale d'amplitude *A* et de fréquence *f*, donnée par l'expression [**plus..**]:

$$y(t) = A\sin(\omega * t) \text{ where } \omega = 2 * \pi * f$$

Un signal plus complexe, comme une note tenue de manière constante par un instrument, est également périodique mais sans être une sinusoïde. Le célèbre mathématicien Joseph Fourier (1768-1830) a prouvé qu'il était toujours possible de représenter une fonction périodique quelconque par une somme d'oscillations sinusoïdales, appelée **série de Fourier**. Cette théorie constitue de ce fait un fondement primordial du développement des mathématiques modernes, de la physique et des sciences de l'ingénieur. L'**analyse spectrale** consiste à décomposer un signal en ses composantes sinusoïdales.

CONCEPTS DE PROGRAMMATION: *Oscillation sinusoïdale, Séries de Fourier, Transformation de Fourier rapide (FFT = Fast Fourier Transform), spectre, sonogramme*

SPECTRE D'UN SON, HARMONIQUES

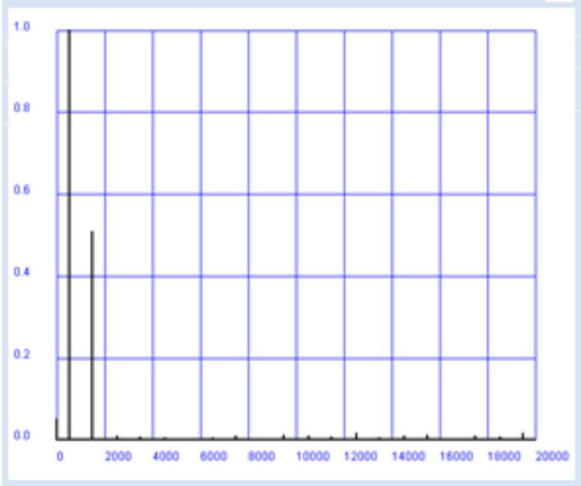
Les composantes de fréquences sinusoïdales déterminent le timbre sonore d'une voix ou d'un instrument de musique. Un son purement périodique est constitué de la fondamentale et de ses harmoniques dont les fréquences sont des multiples entiers de la fondamentale. Si l'on représente graphiquement l'amplitude de chacune des différentes composantes de fréquences du son, on obtient son **spectre** que l'on peut déterminer. Un appareil permettant de déterminer le spectre d'un son est appelé **analyseur de spectre**. *TigerJython* est capable de déterminer le spectre d'un son à l'aide d'un célèbre algorithme appelé transformation de Fourier rapide (FFT = ***Fast Fourier Transform***).

Pour effectuer une transformation de Fourier rapide, on passe à la fonction *fft(samples, n)* une liste contenant les valeurs échantillonnées à intervalles de temps réguliers. Le paramètre *n* permet d'indiquer que l'on ne prend que les *n* premières valeurs de la liste pour effectuer la FFT.

La fonction *fft* retourne une liste de nombres représentant l'amplitude de chacune des*n/2* composantes de fréquences (normalisées) du son analysé. Chaque nombre ainsi obtenu par la fonction *fft* représente une fréquence présente dans le spectre du son analysé. La différence de fréquence entre chaque composante retournée est de *r = fs / n,* où *fs est la fréquence d'échantillonnage.* La grandeur *r* est appelée **résolution** du spectre.

Ces *n/2* valeurs de retour espacées d'une différence de fréquence *r* couvrent la plage de fréquence entre 0 et *n/2*r = fs/2*, ou, pour faire simple : **La FFT permet de déterminer le spectre compris entre *0* et *fs/2* pour une fréquence d'échantillonnage de *fs***. Pour prendre un exemple concret, l'échantillonnage à 44100 Hz d'un CD audio permet de couvrir une plage de fréquence de 0 à 22050 Hz, ce qui correspond à l'ensemble de la page de fréquence audibles par l'oreille humaine.

Afin de tester notre analyseur de spectre, nous allons commencer par utiliser le son *"wav/doublesine.wav"* présent dans la distribution de TigerJython. Celui-ci superpose deux sons sinusoïdaux et a été enregistré à une fréquence d'échantillonnage *fs* = 40,000 Hz. En prenant *n* = 10,000 valeurs d'échantillonnage, la fonction *fft(samples, n)* retourne 5'000 composantes de fréquences avec une résolution de *r* = 40,000 / 10,000 = 4 Hz dans le domaine de fréquence entre 0 et 20,000 Hz que l'on peut ensuite représenter à l'aide de barres verticales dans un *GPanel*.



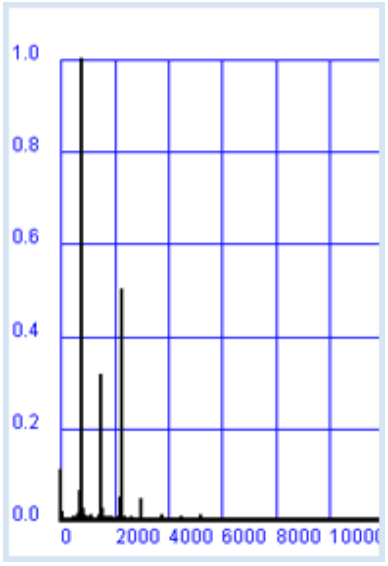

```
from soundsystem import *
from gpanel import *

def showSpectrum(text):
    makeGPanel(-2000, 22000, -0.2, 1.2)
    drawGrid(0, 20000, 0, 1.0, 10, 5, "blue")
    title(text)
    lineWidth(2)
    r = fs / n # Resolution
    f = 0
    for i in range(n // 2):
        line(f, 0, f, a[i])
        f += r

fs = 40000 # Sampling frequency
n = 10000 # Number of samples
samples = getWavMono("wav/doublesine.wav")
openMonoPlayer(samples, fs)
play()
a = fft(samples, n)
showSpectrum("Audio Spectrum")
```

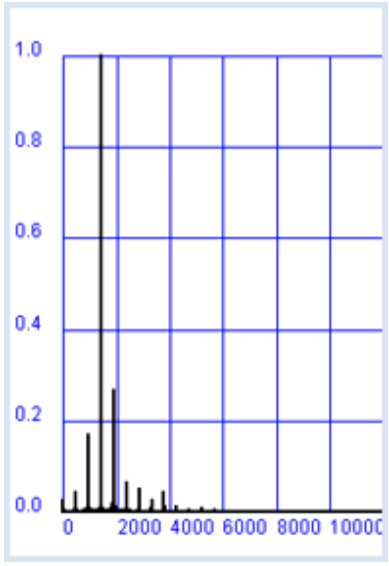
Hihliht proram code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Comme vous pouvez l'imaginer et l'entendre, le spectre contient deux composantes de fréquences différentes : 500 Hz et 1.5 kHz présentant un rapport d'amplitude de 1 pour 1/2. L'analyseur de spectre révèle également quelques autres composantes de fréquences perturbatrices. La fréquence 0 correspond à une composante de signal constante (offset).

On dispose à présent d'un analyseur de spectre permettant d'examiner les fondamentales et les harmoniques de divers instruments de musiques, de voix humaines ou animales. On trouve un enregistrement de flûte ("wav/flute.wav") et de hautbois ("wav/oboe.wav") tout prêts dans la distribution de TigerJython et l'on voit immédiatement qu'ils présentent des caractéristiques sonores très différentes.

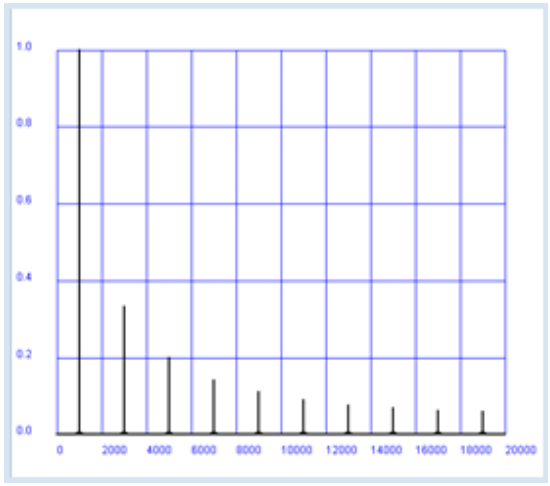




SPECTRES DE FONCTIONS QUELCONQUES

D'après le théorème de Fourier, toute fonction périodique de fréquence *f* peut être représentée comme une superposition de fonctions sinusoïdales de fréquences f, 2*f, 3*f, etc. (Séries de Fourier).

Nous pouvons sans problème déterminer expérimentalement l'amplitude des composantes de fréquences avec notre analyseur de Fourier. Dans le cas présent, on considère une onde carrée de fréquence *f* = 1 kHz. La fonction prédéfinie *square(A, f, t)* retourne la valeur *A* durant la première moitié de la période et la valeur *-A* durant la deuxième moitié.



Dans le programme suivant, on choisit une fréquence d'échantillonnage de *fs* = 40 kHz et l'on détermine les échantillons sonores sur une durée de 3 secondes (120'000 valeurs). On joue ensuite le son capturé et on n'utilise que 10'000 valeurs pour afficher le spectre.

```
from soundsystem import *
from gpanel import *

def showSpectrum(text):
    makeGPanel(-2000, 22000, -0.2, 1.2)
    drawGrid(0, 20000, 0, 1.0, 10, 5, "blue")
    title(text)
    lineWidth(2)
    r = fs / n # Resolution
    f = 0
    for i in range(n // 2):
        line(f, 0, f, a[i])
        f += r

n = 10000
fs = 40000 # Sampling frequency
f = 1000 # Signal frequency

samples = [0] * 120000  # sampled data for 3 s
t = 0
```

```
dt = 1 / fs # sampling period
for i in range(120000):
    samples[i] = square(1000, f, t)
    t += dt

openMonoPlayer(samples, 40000)
play()
a = fft(samples, n)
showSpectrum("Spectrum Square Wave")
```

Hihliht proram code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

L'expérience montre que le spectre d'une fonction rectangulaire est constitué des multiples impairs de la fréquence fondamentale et que les amplitudes de ces composantes de fréquences sont les termes de la suite 1, 1/3, 1/5, 1/7, etc. Il n'est cependant pas possible de déterminer par l'expérience que les composantes spectrales s'étendent à l'infini d'un point de vue théorique.

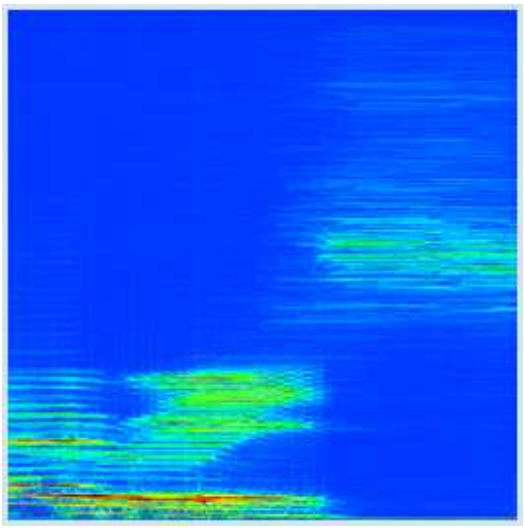
SONOGRAMME

La transformation de Fourier rapide est un outil idéal pour enregistrer le comportement spectral d'un son variant dans le temps tel qu'un mot parlé. Bien entendu, le signal n'est dans ce cas plus périodique mais on peut faire l'hypothèse qu'il est périodique par morceaux. C'est la raison pour laquelle l'algorithme FFT est souvent utilisé, à intervalles réguliers de 2.5 ms, sur de courts blocs de signaux de 100 ms par exemple. On obtient de ce fait un nouveau spectre toutes les 2.5 ms qui peut être représenté par des bandes verticales colorées dans un sonogramme.

In your proram, you start at the beinnin of the samplin values and analyze a block lenth of 2000 values. You bein the next block 50 samples later, etc. In Python, you can do this with a slice operation

samples[k * 50:] where k = 0, 1, 2,...

This results in a **sonoram,** for example for the spoken word "harris", located in the distribution of *TierJython* as "*wav/harris.wav*".



```
from soundsystem import *
from gpanel import *

def toColor(z):
    w = int(450 +  300 * z)
    c = X11Color.wavelengthToColor(w)
    return c

def drawSonogram():
    makeGPanel(0, 190, 0, 1000)
    title("Sonogramm of 'Harris'")
    lineWidth(4)
    # Analyse blocks every 50 samples
    for k in range(191):
```

```
        a = fft(samples[k * 50:], n)
        for i in range(n // 2):
            setColor(toColor(a[i]))
            point(k, i)

fs = 20000 # Sampling freq->spectrum 0..10 kHz
n = 2000 #  Size of block for analyser

samples = getWavMono("wav/harris.wav")
openMonoPlayer(samples, fs)
play()
drawSonogram()
```

Hihliht proram code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

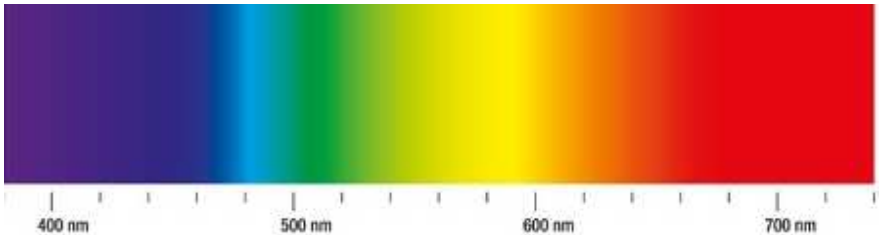
Le sonogramme produit présente sur l'axe vertical les fréquences entre 0 et 10 kHz et, horizontalement, l'évolution temporelle du spectre de 0 à 190 * 50 / 20000 = 0.475 s.

Pour effectuer la conversion entre les valeurs numériques et les couleurs, on utilise la fonction *X11Color.wavelengthToColor*() qui permet de convertir des longueurs d'onde du spectre visible compris entre 380 et 780 nm en une couleur affichable à l'écran.

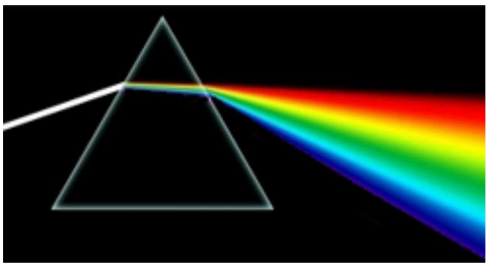
À la fin du mot, lorsque le « s » sifflant est prononcé, les hautes composantes spectrales sont clairement visibles, alors que les fondamentales sont complètement absentes.

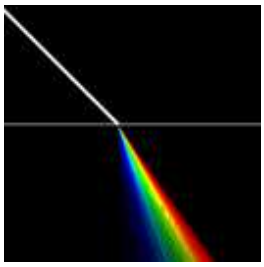
SPECTRE DE LA LUMIÈRE

La lumière peut également être décomposée de manière spectrale pour déterminer la longueur d'onde de ses différentes composantes. Les longueurs d'onde du spectre visible s'étendent entre 380 nm et 780 nm.



Il est fort probable que vous connaissiez déjà l'analyseur de spectre pour la lumière, à savoir le prisme. Celui-ci fonctionne en réfractant davantage les longueurs d'ondes bleues (380 nm) que les rouges (780 nm), selon la loi de la réfraction.





Le programme suivant simule la transition d'un rayon lumineux blanc provenant du vide et pénétrant dans le verre en montrant le chemin emprunté par les différentes couleurs au moyen d'un agrandissement.

```
from gpanel import *

# K5 glass
```

```
B = 1.5220
C = 4590  # nanometer^2
# Cauchy equation for refracting index
def n(wavelength):
    return B + C / (wavelength * wavelength)

makeGPanel(-1, 1, -1, 1)
title("Refracting at the K5 glass")
bgColor("black")
setColor("white")
line(-1, 0, 1, 0)


lineWidth(4)
line(-1, 1, 0, 0)
lineWidth(1)

sineAlpha = 0.707

for i in range(51):
     wavelength = 380 + 8 * i
     setColor(X11Color.wavelengthToColor(wavelength))
     sineBeta = sineAlpha / n(wavelength)
     x = (sineBeta - 0.45) * 100 - 0.5  # magnification
     line(0, 0, x, -1)
```

Hihliht proram code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Afin d'obtenir un beau graphique, la lumière est davantage réfractée que dans la réalité.

EXERCISES

1. Utiliser l'analyseur spectral développé dans ce chapitre pour étudier d'autres instruments ou voix en visualisant leur fondamentale et leurs harmoniques. Essayer d'interpréter les résultats en fonction du caractère sonore de l'instrument en question.

2. En plus de la fonction globale *square(A, f, t),* on dispose également des fonctions *sine(a, f, t), triangle(A, f, t) et sawtooth(A, f, t)*. Tentez de prédire l'aspect du spectre d'une onde triangulaire ou d'une onde en dent de scie. Vous pouvez également analyser la superposition d'ondes sinusoïdales générées à l'aide de la fonction *sine()*.

3. À l'aide de sonogrammes, analyser les différences que présentent différentes voix féminines et masculines prononçant le même mot.

8.9 DYNAMIQUE DE GROUPE

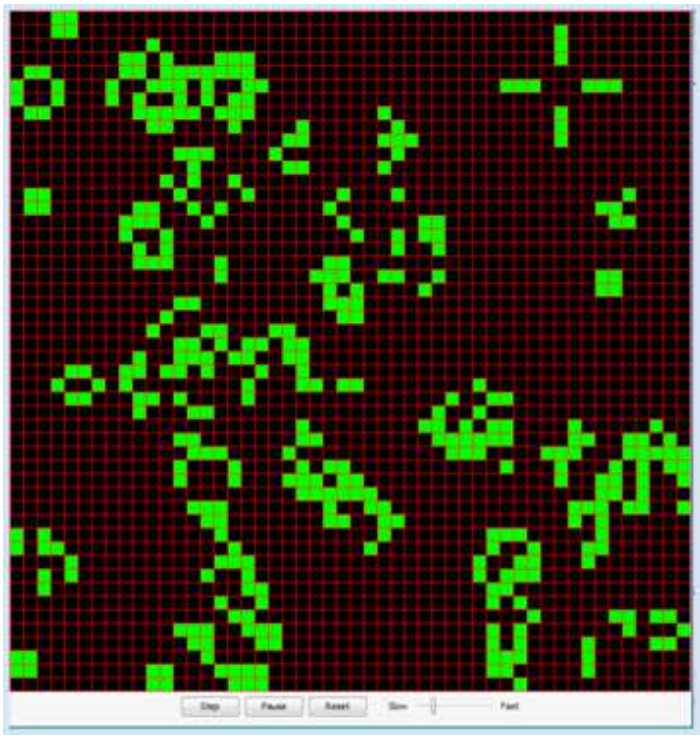
INTRODUCTION

Les systèmes comportant de nombreux partenaires en interaction sont très répandus. Il est souvent possible de simuler sans problème de tels systèmes à l'aide de programmes informatiques puisque l'ordinateur peut sans autre stocker des milliers ou même des millions d'états individuels tout en suivant leur évolution temporelle. Il va sans dire qu'une telle simulation n'est pas possible avec des systèmes à l'échelle atomique comportant un nombre de particules en interaction de l'ordre de 1023. Ceux-ci dépassent en effet de loin les capacités de stockage et de traitement des ordinateurs. Pour simuler de tels systèmes, il faut utiliser des procédures simplifiées consistant par exemple à diviser le système en plusieurs cellules plus larges. Les simulations des phénomènes liés à l'atmosphère terrestre utilisées pour effectuer les prévisions météorologiques ou les prédictions de l'évolution du climat sur le long terme sont de bons exemples de telles simplifications.

CONCEPTS DE PROGRAMMATION: *Simulation informatique, dynamique de population, comportement en essaim*

LE JEU DE LA VIE DE CONWAY

Le jeu de la vie de Conway a pour but d'étudier le comportement d'individus en interaction disposés dans une structure de grille bidimensionnelle. Chaque individu est en interaction avec les huit cases qui lui sont adjacentes. Cette simulation fut proposée par le mathématicien britannique John Conway en 1970 et le rendit célèbre même en dehors des cercles mathématiques. Pratiquement tout scientifique a au moins une vague idée du jeu de la vie. Vous aurez une idée bien précise de son fonctionnement après l'avoir développé en *Python*.



La population est formée de cellules et évolue dans le temps par étapes discrètes (générations). Chaque cellule peut être en *état de vie* ou en *état de mort*.

La simulation passe de la génération *i* à la génération *i+1* en considérant, pour chaque cellule, les règles de transitions suivantes basées sur l'état des huit cellules environnantes :

1. Si la cellule est vivante en *i*, elle va mourir si elle comporte moins d'une cellule voisine vivante actuellement (isolation)
2. Si la cellule est vivante en *i*, elle va continuer à vivre si elle possède deux ou trois cellules actuellement vivantes (cohésion de groupe)

3. Si la cellule est vivante en *i*, elle va mourir si elle possède plus de trois cellules voisines vivantes (surpopulation)
4. Si une cellule est morte en *i*, elle revient à la vie si elle possède exactement trois cellules voisines vivantes en *i,* (reproduction). Sinon, elle reste morte.

La structure en grille de *GameGrid* convient à merveille pour implémenter ce jeu. On utilise une liste bidimensionnelle *a[x][y]* pour encoder l'état de chaque cellule de la population en représentant une cellule morte par un 0 et une cellule vivante par un 1. Le cycle de simulation de *GameGrid* est considéré comme le temps de vie d'une génération. La population actuelle stockée dans la liste a est copiée dans la liste b au sein de la fonction de rappel *onAct()* et sera finalement considérée comme la liste actuelle. La population est initialisée en choisissant au hasard 1000 cellules initialement vivantes dans la fonction de rappel *onReset()* qui est appelée lors d'un clic sur le bouton « reset ».
Il ne faut pas oublier d'enregistrer les fonctions de rappel à l'aide des fonctions *registerAct()* et *registerNavigation().*

```
from gamegrid import *

def onReset():
    for x in range(s):
        for y in range(s):
            a[x][y] = 0  # All cells dead
    for n in range(z):
        loc = getRandomEmptyLocation()
        a[loc.x][loc.y] = 1
    showPopulation()

def showPopulation():
    for x in range(s):
        for y in range(s):
            loc = Location(x, y)
            if a[x][y] == 1:
                getBg().fillCell(loc, Color.green, False)
            else:
                getBg().fillCell(loc, Color.black, False)
    refresh()

def getNumberOfNeighbours(x, y):
    nb = 0
    for i in range(max(0, x - 1), min(s, x + 2)):
        for k in range(max(0, y - 1), min(s, y + 2)):
            if not (i == x and k == y):
                if a[i][k] == 1:
                    nb = nb + 1
    return nb

def onAct():
    global a
    # Don't use the current, but a new population
    b  = [[0 for x in range(s)] for y in range(s)]
    for x in range(s):
        for y in range(s):
            nb = getNumberOfNeighbours(x, y)
            if a[x][y] == 1:  # living cell
                if nb < 2:
                    b[x][y] = 0
                elif nb > 3:
                    b[x][y] = 0
                else:
                    b[x][y] = 1
            else:             # dead cell
                if nb == 3:
                    b[x][y] = 1
                else:
                    b[x][y] = 0
```

```
    a = b # Use new population as current
    showPopulation()

# ================== global section ==================
s = 50   # Number of cells in each direction
z = 1000 # Size of population at start
a  = [[0 for x in range(s)] for y in range(s)]
makeGameGrid(s, s, 800 // s, Color.red)
registerAct(onAct)
registerNavigation(resetted = onReset)
setTitle("Conway's Game Of Life")
onReset()
show()
```

S&ecute;lectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

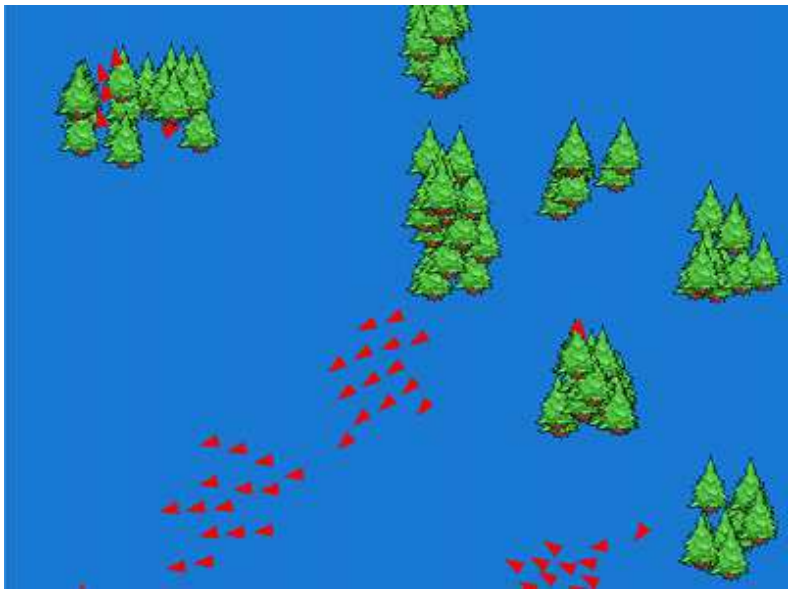
Le jeu de la vie est un exemple typique d'**automate cellulaire** constitué de cellules d'une grille en interaction les unes avec les autres. Les automates cellulaires se prêtent très bien à l'étude du comportement de systèmes naturels complexes parmi lesquels on trouve

- La croissance biologique, l'émergence de la vie
- Le comportement social, géologique, écologique
- La régulation du trafic
- La formation et l'évolution du cosmos, des galaxies et des étoiles

En 2002, Stefan Wolfram, le scientifique et développeur en chef de Mathematica fit remarquer, dans son fameux livre "A New Kind of Science", que de tels systèmes peuvent être étudiés à l'aide de programmes simples. Les simulations informatiques nous placent à la veille d'une nouvelle ère dans la manière d'acquérir des connaissances scientifiques.

Pour initialiser une liste à deux dimensions (liste de listes), on utilise une syntaxe python spéciale appelée **compréhension de liste (***list comprehension*en anglais**)** (voir **Matériel supplémentaire**).

COMPORTEMENT EN ESSAIM



Comme vous avez déjà pu l'observer souvent dans la vie, de nombreux êtres vivants ont tendance à former de grands groupes d'individus. Ce phénomène est particulièrement évident chez les oiseaux, les

poissons et les insectes. Un groupe de personnes réunies pour une manifestation présentent également une forme de « comportement grégaire ». La formation d'un essaim d'individus est influencée d'une part par des facteurs extérieurs globaux et d'autre part par les interactions des partenaires dans leur environnement immédiat (influences locales).

En 1986, Craig Reynolds a démontré que la formation d'un essaim d'individus qu'il appela *Boids* est régie par les trois règles suivantes:

1. **Règle de cohésion** : se déplacer vers le centre de gravité des individus se trouvant dans son voisinage
2. **Règle de séparation**: s'éloigner si l'on se rapproche trop près d'un autre individu
3. **Règles d'alignement** : se déplacer approximativement dans la même direction que ses voisins

Afin d'implémenter ce comportement, on utilise à nouveau *JGameGrid* afin de minimiser les ressources nécessaires pour effectuer le rendu de l'animation. Il est alors possible d'utiliser une grille de jeu dont les cellules se réduisent à un pixel et de spécifier la position, la vitesse et l'accélération des acteurs à l'aide de vecteurs en virgule flottante de la classe *GGVector*. À chaque période de simulation, on commence par déterminer le nouveau vecteur accélération à partir des trois règles de formation d'essaim grâce à la fonction *setAcceleration()*. On obtient ainsi les nouveaux vecteurs vitesse et position

$$\vec{v}' = \vec{v} + \vec{a} * dt \qquad \text{et} \qquad \vec{r}' = \vec{r} + \vec{v} * dt$$

Puisque l'échelle de temps absolu n'est pas essentielle, on peut sans autre fixer l'incrément temporel à $dt = 1$.

L'application de la règle de séparation conduit non seulement à une répulsion entre les oiseaux trop rapprochés, mais également un évitement des obstacles, en l'occurrence les arbres.

Il faut faire particulièrement attention à bien gérer les limites de la zone de vol (les côtés du canevas). On peut imaginer plusieurs possibilités. On pourrait, par exemple, utiliser une topologie toroïdale dans laquelle les oiseaux quittant la zone de vol d'un côté réapparaissent automatiquement de l'autre. Dans le programme suivant, on a simplement fait le choix d'imposer aux oiseaux de faire demi-tour lorsqu'ils parviennent aux bords.

```
from gamegrid import *
import math
import random

# ================== class Tree ======================
class Tree(Actor):
    def __init__(self):
        Actor.__init__(self, "sprites/tree1.png")

# ================== class Bird ======================
class Bird(Actor):
    def __init__(self):
        Actor.__init__(self, True, "sprites/arrow1.png")
        self.r = GGVector(0, 0)  # Position
        self.v = GGVector(0, 0)  # Velocity
        self.a = GGVector(0, 0)  # Acceleration

    # Called when actor is added to gamegrid
    def reset(self):
        self.r.x = self.getX()
        self.r.y = self.getY()
        self.v.x = startVelocity * math.cos(math.radians(self.getDirection()))
        self.v.y = startVelocity * math.sin(math.radians(self.getDirection()))

    # ---------- cohesion ---------------
    def cohesion(self, distance):
        return self.getCenterOfMass(distance).sub(self.r)
```

```
    # ----------- alignment --------------
    def alignment(self, distance):
        align = self.getAverageVelocity(distance)
        align = align.sub(self.v)
        return align

    # ----------- separation -------------
    def separation(self, distance):
        repulse = GGVector()
        # ------ from birds ------
        for p in birdPositions:
            dist = p.sub(self.r)
            d = dist.magnitude()
            if d < distance and d != 0:
                repulse = repulse.add(dist.mult((d - distance) / d))

        # ------ from trees ------
        trees = self.gameGrid.getActors(Tree)
        for actor in trees:
            p = GGVector(actor.getX(), actor.getY())
            dist = p.sub(self.r)
            d = dist.magnitude()
            if d < distance and d != 0:
               repulse = repulse.add(dist.mult((d - distance) / d))
        return repulse

    # ----------- wall interaction -------
    def wallInteraction(self):
        width = self.gameGrid.getWidth()
        height = self.gameGrid.getHeight()
        acc = GGVector()
        if self.r.x < wallDist:
            distFactor = (wallDist - self.r.x) / wallDist
            acc = GGVector(wallWeight * distFactor, 0)
        if width - self.r.x < wallDist:
            distFactor = ((width - self.r.x) - wallDist) / wallDist
            acc = GGVector(wallWeight * distFactor, 0)
        if self.r.y < wallDist:
            distFactor = (wallDist - self.r.y) / wallDist
            acc = GGVector(0, wallWeight * distFactor)
        if height - self.r.y < wallDist:
            distFactor = ((height - self.r.y) - wallDist) / wallDist
            acc = GGVector(0, wallWeight * distFactor)
        return acc

    def getPosition(self):
        return self.r

    def getVelocity(self):
        return self.v

    def getCenterOfMass(self, distance):
        center = GGVector()
        sum = 0
        for p in birdPositions:
            dist = p.sub(self.r)
            d = dist.magnitude()
            if d < distance:
                center = center.add(p)
                sum += 1
        if sum != 0:
            return center.mult(1.0/sum)
        else:
            return center

    def getAverageVelocity(self, distance):
        avg = GGVector()
        sum = 0
```

```
        for i in range(len(birdPositions)):
            p = birdPositions[i]
            if (self.r.x - p.x) * (self.r.x - p.x) + \
               (self.r.y - p.y) * (self.r.y - p.y) < distance * distance:
                avg = avg.add(birdVelocities[i]);
                sum += 1
        return avg.mult(1.0/sum)

    def limitSpeed(self):
        m = self.v.magnitude()
        if m < minSpeed:
            self.v = self.v.mult(minSpeed / m)
        if m > maxSpeed:
            self.v = self.v.mult(minSpeed / m)

    def setAcceleration(self):
        self.a = self.cohesion(cohesionDist).mult(cohesionWeight)
        self.a = self.a.add(self.separation(separationDist).mult(separationWeight))
        self.a = self.a.add(self.alignment(alignmentDist).mult(alignmentWeight))
        self.a = self.a.add(self.wallInteraction())

    def act(self):
        self.setAcceleration()
        self.v = self.v.add(self.a) # new velocity
        self.limitSpeed()
        self.r = self.r.add(self.v) # new position
        self.setDirection(int(math.degrees(self.v.getDirection())))
        self.setLocation(Location(int(self.r.x), int(self.r.y)))


# ================== global section ==================
def populateTrees(number):
    blockSize = 70
    treesPerBlock = 10
    for block in range(number // treesPerBlock):
        x = getRandomNumber(800 // blockSize) * blockSize
        y = getRandomNumber(600 // blockSize) * blockSize
        for t in range(treesPerBlock):
            dx = getRandomNumber(blockSize)
            dy = getRandomNumber(blockSize)
            addActor(Tree(), Location(x + dx, y + dy))

def generateBirds(number):
    for i in range(number):
        addActorNoRefresh(Bird(), getRandomLocation(),
                                  getRandomDirection())
    onAct()  # Initialize birdPositions, birdVelocities

def onAct():
    global birdPositions, birdVelocities
    # Update bird positions and velocities
    birdPositions = []
    birdVelocities = []
    for b in getActors(Bird):
        birdPositions.append(b.getPosition())
        birdVelocities.append(b.getVelocity())

def getRandomNumber(limit):
    return random.randint(0, limit-1)

# coupling constants
cohesionDist = 100
cohesionWeight = 0.01
alignmentDist = 30
alignmentWeight = 1
separationDist = 30
separationWeight = 0.2
wallDist = 20
wallWeight = 2
```

```
maxSpeed = 20
minSpeed = 10
startVelocity = 10
numberTrees = 100
numberBirds = 50

birdPositions  = []
birdVelocities  = []

makeGameGrid(800, 600, 1, False)
registerAct(onAct)
setSimulationPeriod(10)
setBgColor(makeColor(25, 121, 212))
setTitle("Swarm Simulation")
show()
populateTrees(numberTrees)
generateBirds(numberBirds)
doRun()
```

S&ecute;lectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

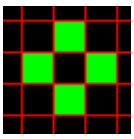
MEMENTO

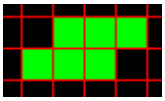
La simulation dépend d'un certain nombre de **constantes de couplage** qui déterminent l'intensité de l'interaction entre les individus. Ces valeurs sont très sensibles et il pourrait être nécessaire de les ajuster en fonction de la puissance de traitement de votre ordinateur.

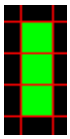
Encore une fois, la fonction de rappel *onAct()* est activée en utilisant *registerAct()* de sorte qu'elle est appelée automatiquement à chaque cycle de simulation. Le mouvement des oiseaux est géré par la méthode *act()* de la classe *Bird*.

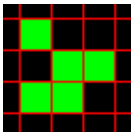
EXERCICES

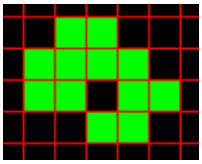
1. Étudier le comportement des motifs initiaux suivants dans le jeu de la vie

. 

b. 

c. 

d. 

e. 

2. Décrire trois comportements en essaim typiques survenant dans le monde animal. Pour chaque exemple, réfléchissez aux raisons qui amènent ces animaux à former des essaims.

3*. Dans la simulation avec les oiseaux, introduire trois oiseaux prédateurs qui pourchassent les volées d'oiseaux et qui sont évités par les proies. Consigne : utiliser l'image de sprite *arrow2.png* pour représenter les prédateurs.

4*. Modifier la simulation de telle sorte que les prédateurs de l'exercice précédent mangent les oiseaux proies lors d'une collision.

MATÉRIEL SUPPLÉMENTAIRE

COMPRÉHENSION DE LISTE

En Python, il est possible de créer des listes de manière très élégante en utilisation une syntaxe spéciale inspirée de la notation de la théorie des ensembles que vous connaissez bien des mathématiques :

Mathématiques	*Python*
S = {x: x élément de {1...10}}	s = [x for x in range(1, 11)]
T = {x^2: x élément de {0...10}}	t = [x**2 for x in range(11)]
V = {x | x élément de S et x pair}	v = [x for x in s if x % 2 == 0]
$M = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$	m = [[0 for x in range(3)] for y in range(3)]

N'hésitez pas à utiliser la console pour tester les expressions Python. Pour ce faire, vous pouvez les copier depuis le tableau ci-dessous et les coller dans la console:

```
>>> s = [x for x in range(1, 11)]
>>> s
 < [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> t = [x**2 for x in range(11)]
 < [0, 1, 4, 9, 16, 25, 35, ..., 100]
>>> v = [x for x in s if x% 2 == 0]
 < [2, 4, 6, 8, 10]
>>> m = [[0 for x in range(3)] for y in range(3)]
>>> m
 < [[0, 0, 0],
    [0, 0, 0],
    [0, 0, 0]]
```

8.10 MARCHE ALÉATOIRE

INTRODUCTION

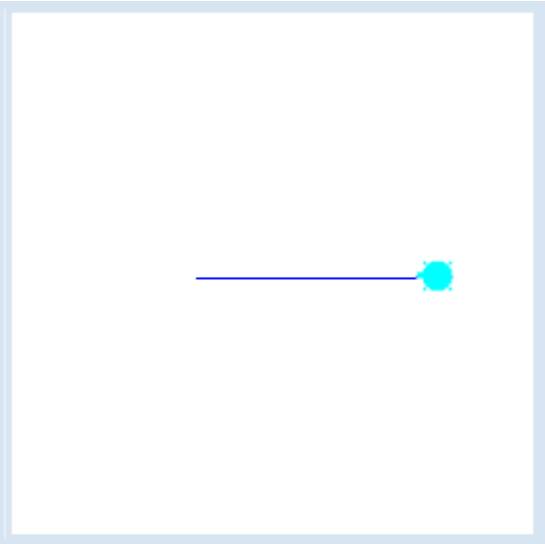
Dans la vie quotidienne, de nombreux phénomènes sont déterminés par le hasard et il est souvent nécessaire de prendre des décisions sur la base de probabilités. On pourrait par exemple être amenés à préférer un mode de transport à un autre sur la base de leur probabilité respective d'être impliqué dans un accident. Dans de telles situations, l'ordinateur peut se révéler être un outil important permettant d'examiner les dangers potentiels à l'aide simulations, en ne prenant absolument aucun risque.

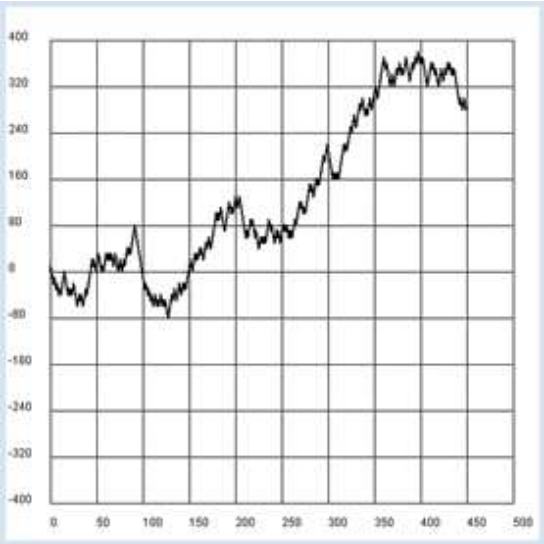
Imaginons qu'une personne se soit perdue et qu'elle ne trouve plus le chemin pour rentrer à la maison. Elle va se mettre à effectuer une marche aléatoire consistant à se déplacer d'une même distance à chaque intervalle de temps, mais en choisissant une direction aléatoire. Même si un tel mouvement ne correspond pas nécessairement à la réalité, il permet de découvrir des caractéristiques importantes applicables à des systèmes réels comme la modélisation de marchés boursiers en mathématiques financières ou du déplacement de molécules.

CONCEPTS DE PROGRAMMATION: Marche aléatoire, mouvement brownien

MACHE ALÉATOIRE À UNE DIMENSION

Une fois encore, les graphiques tortues sont très utiles pour représenter la simulation graphiquement. Au temps 0, la tortue se trouve à la position x = 0 et se déplace le long de l'axe Ox par pas d'égale longueur. À chaque pas, la tortue "décide" de faire un pas vers la gauche ou vers la droite de manière aléatoire. Dans le programme suivant, les deux choix surviennent avec la même probabilité p = ½ (marche aléatoire symétrique).






```
from gturtle import *
from gpanel import *
import random

makeTurtle()
makeGPanel(-50, 550, -480, 480)
```

```
windowPosition(880, 10)
drawGrid(0, 500, -400, 400)
title("Mean distance versus time")
lineWidth(2)
setTitle("Random Walk")

t = 0
while t < 500:
    if random.randint(0, 1) == 1:
        setHeading(90)
    else:
        setHeading(-90)
    forward(10)
    x = getX()
    draw(t, x)
    t += 1
print "All done"
```

MEMENTO

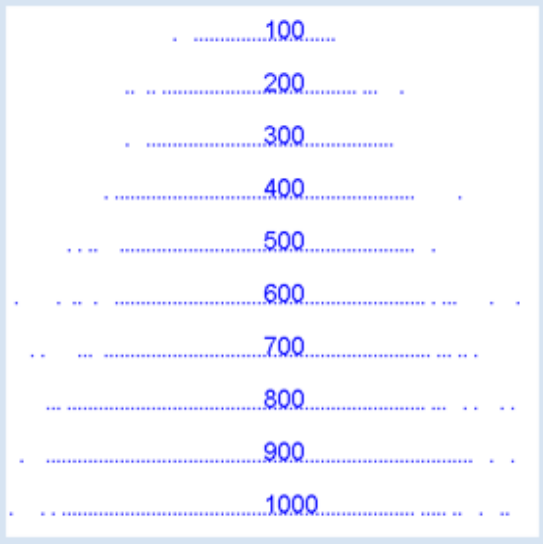
Vous êtes peut-être surpris d'observer que, dans la plupart des cas, la tortue s'éloigne peu à peu de son point de départ alors même qu'elle fait, à chaque étape, un pas de même longueur à gauche ou à droite avec la même probabilité. Afin d'examiner ce résultat de manière plus minutieuse, notre prochain programme va déterminer la distance moyenne séparant la tortue du point de départ après t étapes parmi 1'000 marches aléatoires.

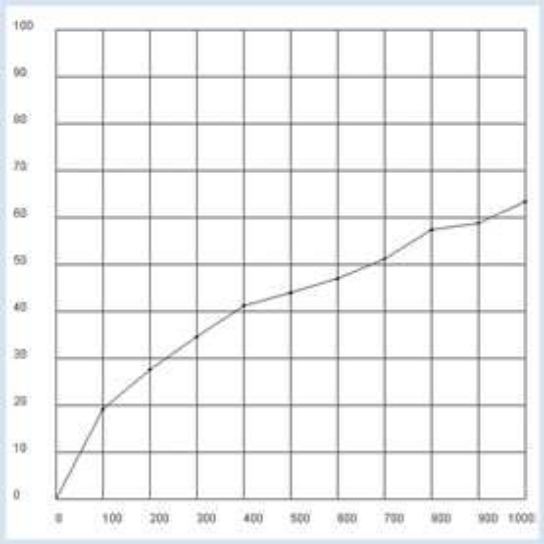
LA LOI DE LA RACINE CARRÉE DU TEMPS

Le programme ci-dessous permet de visualiser de quelle distance une marche aléatoire de t pas s'éloigne de l'origine en moyenne pour t valant 100, 200, 300, …, 1000. Pour chaque nombre de pas t, la simulation est répétée 1000 fois (1000 marches aléatoires) dont le point d'arrivée est représenté à l'écran. La coordonnée y correspond au nombre de pas que comporte la marche aléatoire. Afin d'accélérer l'obtention des résultats, il est conseillé de cacher la tortue et d'éviter d'en dessiner la trace. Chacune de ces simulations permet de déterminer une distance r séparant le point final du point initial. Pour chaque jeu d'expériences avec un t donné, on obtiendra ainsi une distance d'éloignement moyenne avec le point de départ qui est reportée dans un graphique GPanel muni d'un repère orthonormé.

```
from gturtle import *
from gpanel import *
import math
import random

makeTurtle()
makeGPanel(-100, 1100, -10, 110)
windowPosition(850, 10)
drawGrid(0, 1000, 0, 100)
title("Mean distance versus time")
ht()
pu()

for t in range(100, 1100, 100):
    setY(250 - t / 2)
    label(str(t))
    sum = 0
    repeat 1000:
        repeat t:
            if random.randint(0, 1) == 1:
                setHeading(90)
            else:
```

```
                setHeading(-90)
            forward(2.5)
        dot(3)
        r = abs(getX())
        sum += r
        setX(0)
    d = sum / 1000
    print "t =", t, "d =", d, "q = d / sqrt(t) =", d / math.sqrt(t)
    draw(t, d)
    fillCircle(5)
print "all done"
```





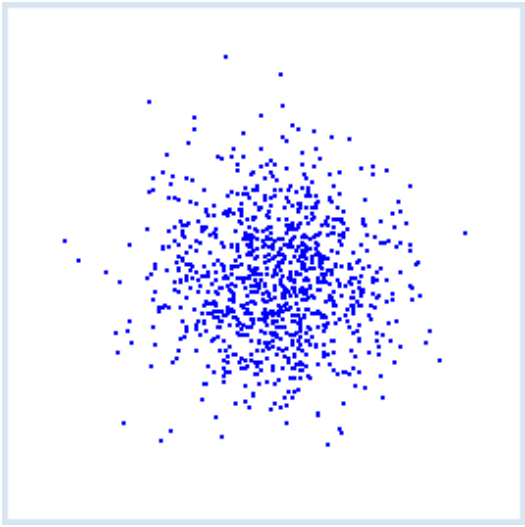
MEMENTO

Comme la visualisation graphique le montre, la distance moyenne séparant le point d'arrivée du point de départ croît avec le nombre de pas de la marche aléatoire. Dans la console, on calcule également le quotient `q = d / sqrt(t)`. Du fait qu'il est toujours pratiquement constant, on peut raisonnablement supposer que la relation `d = q * sqrt(t)` est vraie :

La distance moyenne entre le point de départ et le point d'arrivée de la marche aléatoire augmente selon la racine carrée du temps (nombre d'étapes).

LA MARCHE DU MEC BOURRÉ

Les choses deviennent encore plus intéressantes lorsque la tortue peut se déplacer dans deux dimensions. Elle effectue alors une marche aléatoire à deux dimensions. La tortue effectue des pas de longueur identique mais dans des directions aléatoires.Les mathématiciens et physiciens évoquent souvent ce problème par une histoire à ne pas prendre trop au sérieux.



"Un homme rond comme une barrique tente de regagner son domicile après une longue tournée de bars. Comme il est complètement désorienté, il fait chaque nouveau pas dans une direction aléatoire. En moyenne, à quelle distance se trouve-t-il du bar pour un nombre

de pas donné ? "

Il suffit de modifier l égèrement le programme précédent. On examine à nouveau comment la distance moyenne dépend du temps (du nombre de pas effectués).

```
from gturtle import *
from gpanel import *
import math

makeTurtle()
makeGPanel(-100, 1100, -10, 110)
windowPosition(850, 10)
drawGrid(0, 1000, 0, 100)
title("Mean distance versus time")
ht()
pu()

for t in range(100, 1100, 100):
    sum = 0
    clean()
    repeat 1000:
        repeat t:
           fd(2.5)
           setRandomHeading()
        dot(3)
        r = math.sqrt(getX() * getX() + getY() * getY())
        sum += r
        home()
    d = sum / 1000
    print "t =", t, "d =", d, "q = d / sqrt(t) =", d / math.sqrt(t)
    draw(t, d)
    fillCircle(5)
    delay(2000)
print "all done"
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

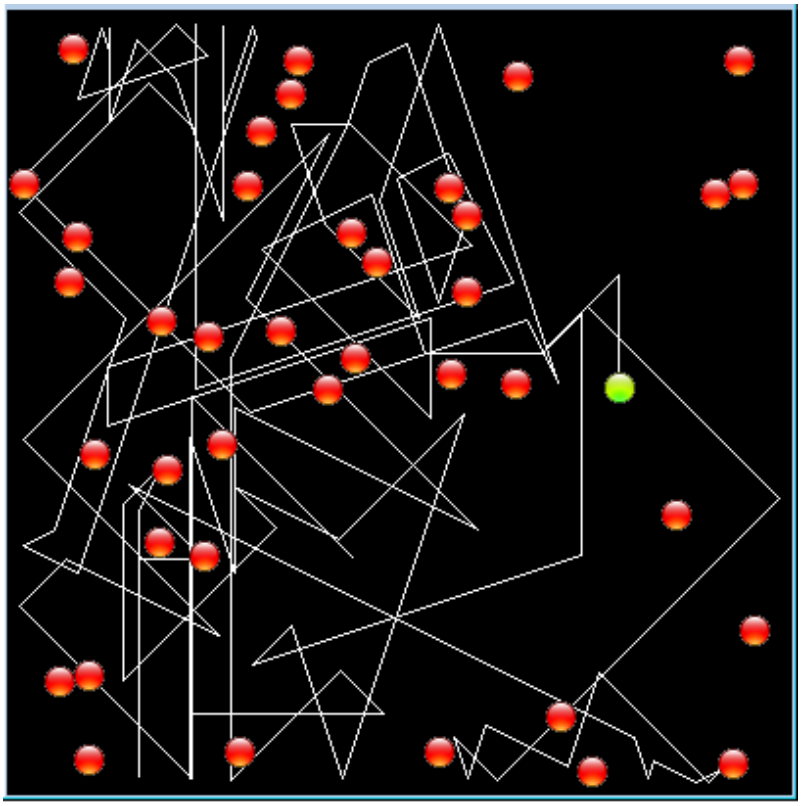
La loi de la racine carrée du temps s'applique également aux marches aléatoires à deux dimensions. En 1905, Albert Einstein en personne a démontré cette loi pour les particules de gaz dans son fameux article « Über die von der molekularkinetischen Theorie der Wärme geforderte Bewegung von in ruhenden Flüssigkeiten suspendierten Teilchen » qui fournit en passant une explication théorique au phénomène du mouvement Brownien. Une année plus tard, M. Smoluchowski est parvenu au même résultat en utilisant une idée différente.

MOUVEMENT BROWNIEN

En 1827 déjà, le biologiste Robert Brown observa au microscope des grains de pollen en suspension dans une goutte d'eau effectuer des mouvements irréguliers de secousses. Il a interprété cette observation en attribuant une force vitale inhérente aux grains de pollen. Il a fallu attendre la découverte de la structure moléculaire de la matière pour pouvoir attribuer ce phénomène au mouvement thermique des molécules d'eau entrant en collision avec les grains de pollen.

Le mouvement brownien peut être joliment montré dans une simulation informatique modélisant les molécules comme de petites sphères en mouvement qui échangent leur vitesse lors des collisions. Cette simulation peut être facilement implémentée à l'aide de JGameGrid puisqu'il est possible de gérer les collisions avec les événements. Pour ce faire, on crée une classe CollisionListener qui dérive de GGActorCollisionListener et on implémente le gestionnaire

d'événements en redéfinissant la méthode collide(). On ajoute chaque particule à l'auditeur (listener en anglais) en utilisant addActorCollisionListener(). Il est possible de régler le type et la taille de la zone de collision à l'aide de la méthode setCollisionCircle(). Dans un souci de simplicité, les 40 particules sont réparties en quatre groupes de vitesses différents.



```
from gamegrid import *

# ==================== class Particle ====================
class Particle(Actor):
    def __init__(self):
        Actor.__init__(self, "sprites/ball.gif", 2)

    # Called when actor is added to gamegrid
    def reset(self):
        self.oldPt = self.gameGrid.toPoint(self.getLocationStart())

    def advance(self, distance):
        pt = self.gameGrid.toPoint(self.getNextMoveLocation())
        dir = self.getDirection()
        # Left/right wall
        if pt.x < 5 or pt.x > w - 5:
            self.setDirection(180 - dir)
        # Top/bottom wall
        if pt.y < 5 or pt.y > h - 5:
            self.setDirection(360 - dir)
        self.move(distance)

    def act(self):
        self.advance(3)
        if self.getIdVisible() == 1:
            pt = self.gameGrid.toPoint(self.getLocation())
            self.getBackground().drawLine(self.oldPt.x, self.oldPt.y, pt.x, pt.y)
            self.oldPt.x = pt.x
            self.oldPt.y = pt.y

# ==================== class CollisionListener =========
class CollisionListener(GGActorCollisionListener):
   # Collision callback: just exchange direction and speed
    def collide(self, a,  b):
```

```
        dir1 = a.getDirection()
        dir2 = b.getDirection()
        sd1 = a.getSlowDown()
        sd2 = b.getSlowDown()
        a.setDirection(dir2)
        a.setSlowDown(sd2)
        b.setDirection(dir1)
        b.setSlowDown(sd1)
        return 10  # Wait a moment until collision is rearmed

# ================== Global section ====================
def init():
    collisionListener = CollisionListener()
    for i in range(nbParticles):
        particles[i] = Particle()
        # Put them at random locations, but apart of each other
        ok = False
        while not ok:
            ok = True
            loc = getRandomLocation()

        for k in range(i):
            dx = particles[k].getLocation().x - loc.x
            dy = particles[k].getLocation().y - loc.y
            if dx * dx + dy * dy < 300:
                ok = False
        addActor(particles[i], loc, getRandomDirection())
        # Select collision area
        particles[i].setCollisionCircle(Point(0, 0), 8)
        # Select collision listener
        particles[i].addActorCollisionListener(collisionListener)
        # Set speed in groups of 10
        if i < 10:
            particles[i].setSlowDown(2)
        elif i < 20:
            particles[i].setSlowDown(3)
        elif i < 30:
            particles[i].setSlowDown(4)
    #  Define collision partners
    for i in range(nbParticles):
        for k in range(i + 1, nbParticles):
            particles[i].addCollisionActor(particles[k])
    particles[0].show(1)

w = 400
h = 400
nbParticles = 40
particles = [0] * nbParticles

makeGameGrid(w, h, 1, False)
setSimulationPeriod(10)
setTitle("Brownian Movement")
show()
init()
doRun()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Les molécules sont modélisées par la classe Particle dérivée de la classe Actor. Elles possèdent deux images de sprite, l'une rouge et l'autre verte, pour permettre de distinguer une molécule particulière dont on voudrait suivre le trajet. L'image verte correspond à spriteID = 1 qui est testé dans la méthode act() pour dessiner la trace avec drawLine().

EXERCICES

1. Une personne effectue une marche aléatoire de 100 pas en partant de x = 0 avec une probabilité p = ½ de marcher vers la droite et une probabilité q = ½ de marcher vers la gauche. Exécuter la simulation 10'000 fois et déterminer la distribution de fréquences de la position finale en l'affichant dans un GPanel.

2. Comme le montre l'exercice 1, la probabilité de se trouver au point de départ à la fin de la marche est la plus élevée. Comment cela peut-il s'accorder avec la loi de la racine carrée du nombre de pas ?

3. Effectuer la même simulation mais en prenant les probabilités p = 0.8 de se déplacer à droite et q = 0.2 de se déplacer à gauche.

4*. Il est possible de prouver que la distribution des fréquences de la position finale correspond à une loi binomiale. La probabilité de se trouver à la coordonnée x après n pas est donnée, pour un nombre de pas n ainsi qu'une position x pairs :

$$P(x, n) = \frac{n!}{((n + x)/2)!((n - x)/2)!} \; p^{(n+x)/2} q^{(n-x)/2}$$

Représenter graphiquement la courbe de la distribution théorique dans l'histogramme obtenu à l'exercice 1.

chapitre neuf

BASES DE DONNÉES & SQL

Objectifs d'apprentissage

- Être capable d'expliquer l'importance des fichiers en informatique.
- Être capable de lire, convertir et sauver des données depuis et vers un fichier.
- Connaître quelques aspects importants des bases de données en ligne et être capable d'installer son propre serveur de base de données.
- Être conscient que, de nos jours, d'immenses quantités de données sont stockées électroniquement dans des bases de données en ligne ainsi que dans les réseaux sociaux.
- Être capable d'utiliser le langage SQL pour générer des tables ainsi que de créer, lire et modifier des jeux de données sur un serveur de base de données.
- Être en mesure de mettre sur pied et de gérer un système de réservation en ligne simple.
- Comprendre le mécanisme de gestion d'exceptions et être capable d'expliquer son importance.

Celui qui traite des données personnelles doit s'assurer qu'elles sont correctes. Il prend toute mesure appropriée permettant d'effacer ou de rectifier les données inexactes ou incomplètes au regard des finalités pour lesquelles elles sont collectées ou traitées.Toute personne concernée peut requérir la rectification des données inexactes.

Suisse. Loi fédérale sur la protection de données,**Article 5**

9.1 PERSISTENCE, FICHIERS

INTRODUCTION

Les informations stockées informatiquement, appelées données, jouent un rôle crucial dans notre société high-tech actuelle. Bien que ces informations soient comparables à du texte écrit, elles comportent également quelques différences notables :

- Les données ne peuvent être lues, stockées et traitées qu'à l'aide d'un **système informatique**
- Les données sont toujours codées **sous forme binaire** à l'aide de 0 et de 1. Elles ne représentent des informations intelligibles que dans la mesure où elles sont correctement interprétées (décodées).
- Les données possèdent une certaine **durée de vie**. Au sein même d'un programme, les données n'ont qu'une durée de vie temporaire, limitée à l'exécution d'un bloc de code en ce qui concerne les variables locales ou à la durée de vie du programme en ce qui concerne les variables globales. Les données persistantes, quant à elles, ont une durée de vie qui transcende l'exécution du programme et peuvent être récupérées après son exécution.
- Les données possèdent une certaine **visibilité (disponibilité)**. Alors que certaines données personnelles peuvent être lues par n'importe qui sur un réseau social, il existe également des données privées ou d'autres types de données figurant sur des supports de données non accessibles au le grand public.
- Les données peuvent être **protégées**. La protection des données peut être réalisée à l'aide **du cryptage** ou de **restrictions d'accès** (protection par mot de passe).
- Les données peuvent être facilement transportées et transférées par **des canaux de communication digitaux**.

Les données persistantes peuvent être écrites ou lues par les programmes informatiques sous la forme de fichiers sur des supports de stockage physiques (Disques durs (HDD = Hard drive disk), disques SSD (= Solid State Drive), cartes mémoires ou clés USB. Un fichier est constitué de zones de stockage au sein d'une certaine structure (système de fichiers) et suivant un **format de données** bien précis. Du fait que le transfert de fichiers est devenu rapide et bon marché même sur de très longues distances, les fichiers sont de nos jours de plus en plus stockés sur des supports distants (**stockage dans le nuage**).

Les fichiers sont gérés sur l'ordinateur au sein d'une structure de dossiers hiérarchique dans laquelle chaque dossier particulier peut contenir non seulement des fichiers mais également des sous-dossiers. Les fichiers sont accessibles à l'aide de leur **chemin d'accès** (*file path* en anglais) contenant le nom de tous les dossiers parents ainsi que le nom du fichier. Le système de fichiers est cependant **dépendant du système d'exploitation**, ce qui explique qu'il existe de grandes différences entre les systèmes Windows, Mac et Linux.

CONCEPTS DE PROGRAMMATION: *Encodage, durée de vie, visibilité de données, fichier*

LIRE ET ÉCRIRE DES FICHIERS TEXTE

Nous avons déjà vu comment lire des fichiers texte dans le chapitre *Internet*. Dans les fichiers texte, les caractères sont stockés de manière séquentielle mais il est possible d'obtenir une structure de fichiers ligne par ligne similaire à celle apparaissant sur une feuille de papier grâce au caractère de fin de ligne *\n*. Et c'est là que le bât blesse, en raison de la différence d'encodage du saut de ligne sur les différents systèmes d'exploitation : alors que les systèmes Mac et Linux utilisent le caractère ASCII <line feed> (EOL), Windows utilise une combinaison de <carriage return><line feed>. En *Python*, ces caractères sont codés à l'aide des séquences d'échappement

\r et \n [**plus...**].

Le programme suivant va utiliser un dictionnaire anglais disponible sous la forme d'un fichier texte qui est téléchargeable (*tfwordlist.zip*) depuis **ce lien.** Il faut décompresser l'archive zip et copier le fichier *words-1$.txt* dans le dossier dans lequel se trouve le programme Python.
Vous allez devoir déterminer, parmi tous les mots contenus dans le dictionnaire en question, lesquels sont des palindromes. Un palindrome est un mot qui se lit exactement de la même manière en allant de gauche à droite ou de droite à gauche, sans considération de la casse (pas de différence entre majuscules ou minuscules).

La fonction *open()* retourne un objet fichier *f*qui permet de manipuler le fichier à proprement parler. Cet objet fichier permet par exemple de parcourir le fichier ligne à ligne à l'aide d'une boucle *for*. Il est de ce fait capital de noter que chaque ligne se termine par un caractère de fin de ligne \n qui doit être ôté par une opération de *slicing* appropriée avant que le mot ne soit lu en sens inverse. De plus, afin de ne pas distinguer les majuscules et minuscules, il faut commencer par convertir tous les caractères en minuscules grâce à la fonction *lower()*.

En Python, il existe une astuce pour retourner une chaîne à l'envers puisque l'opérateur de *slicing* accepte également des indices négatifs qui débutent alors l'indexage de la chaine depuis la fin. Ainsi, si l'on parcourt la chaine avec un pas de -1, la chaine est parcourue à l'envers.

```
def isPalindrom(a):
    return a == a[::-1]

f = open("worte-1$.txt")

print "Searching for palindroms..."
for word in f:
    word = word[:-1] # remove trailing \n
    word = word.lower() # make lowercase
    if isPalindrom(word):
        print word
f.close()
print "All done"
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

La méthode *readline()* permet de lire le fichier ligne à ligne. Il faut s'imaginer un pointeur qui progresse dans le fichier lors de chaque appel de la fonction. Dès que l'on a atteint la fin du fichier, la méthode retourne une chaîne vide. Le programme sauve ensuite la liste des palindromes dans un fichier *palindrom.txt*. Mais avant de pouvoir écrire dans un fichier, il faut l'ouvrir à l'aide de la fonction *open()* en passant la chaine de caractères "w" (pour write = écrire) en tant que second argument. Il est ensuite possible d'écrire vers le fichier à l'aide de la méthode *write()*. Il est impératif de ne pas oublier d'appeler la méthode *close* à la fin de l'écriture pour être certain que l'ensemble des caractères aient été écrits vers le fichier et que le système d'exploitation ait libéré les ressources nécessaires à l'opération.

```
def isPalindrom(a):
    return a == a[::-1]

fInp = open("worte-1$.txt")
fOut = open("palindrom.txt", "w")

print "Searching for palindroms..."
while True:
    word = fInp.readline()
    if word == "":
        break
    word = word[:-1] # remove trailing \n
    word = word.lower() # make lowercase
    if isPalindrom(word):
        print word
        fOut.write(word + "\n")
fInp.close()
```

```
fOut.close()
print "All done"
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Lors de l'ouverture d'un fichier texte avec *open(path, mode)*, le mode utilisateur (user mode), est spécifié sous forme de chaine de caractères grâce au paramètre *mode*.

Mode	Description	Commentaire
"r" (read)	Lecture seule	Le fichier doit exister. Il s'agit de la **valeur par défaut**.
"w" (write)	Ouverture en écriture: le fichier est créé s'il n'existe pas encore et on peut y écrire des données.	Si le fichier existe déjà, son contenu est écrasé.
"a" (append)	Ajoute les données écrites à la fin du fichier.	Le fichier est créé s'il n'existe pas encore.
"r+"	Accès en lecture et en écriture avec concaténation.	Le fichier doit déjà exister.

Pour recommencer la lecture du fichier une fois que le pointeur de lecture est parvenu à la fin du fichier, il faut soit fermer le fichier et le rouvrir ou simplement appeler la méthode *seek(0)* de l'objet fichier retourné par *open()*. Il est également possible de lire en une seule fois tout le contenu du fichier et récupérer son contenu sous forme de chaine de caractères avec

```
text = f.read()
```

sans oublier de fermer le fichier avec *f.close()*. Il est également possible d'obtenir une liste de toutes les lignes contenues dans le fichier, sans les sauts de ligne, avec

```
textList = text.splitlines()
```

Voici encore quelques opérations importantes sur les fichiers :

import os os.path.isfile(path)	Retourne *True* si le fichier *path* existe.
import os os.remove(path)	Supprime le fichier *path*.

SAUVER ET RESTAURER DES OPTIONS ET DES DONNÉES DE JEU

Les fichiers sont fréquemment utilisés pour sauvegarder des informations dans le but de pouvoir les réutiliser lors d'une exécution ultérieure du programme. Il s'agit très souvent de paramètres de configuration apportés par l'utilisateur dans un but de personnalisation. Il se pourrait également que l'on veuille sauver l'état actuel d'un jeu pour pouvoir reprendre la partie ultérieurement en repartant de l'état exact dans lequel le jeu avait été laissé.

En général, les options de configuration et l'état d'un programme sont sauvés de manière très naturelle par des paires clé-valeur où la clé permet d'identifier sans ambiguïté la valeur. L'IDE TigerJython comporte par exemple quelques options de configuration qu'il est utile de connaître:

Clé	Valeur
"autosave"	True
"language"	"de"

Comme nous l'avions vu au chapitre 6.3, il est possible de sauver de telles paires clé-valeur dans un dictionnaire Python que l'on peut ensuite stocker et récupérer sous forme binaire à l'aide du module *pickle*. Avant de fermer la fenêtre, le programme ci-dessous sauvegarde la position et la direction actuelles du crabe ainsi que la position du curseur de vitesse de la simulation. Ces données ainsi sauvées seront lues et restaurées lors du prochain démarrage du jeu.


```
import pickle
import os
from gamegrid import *

class Lobster(Actor):
   def __init__(self):
      Actor.__init__(self, True, "sprites/lobster.gif");

   def act(self):
      self.move()
      if not self.isMoveValid():
          self.turn(90)
          self.move()
          self.turn(90)

makeGameGrid(10, 2, 60, Color.red)
addStatusBar(30)
show()

path = "lobstergame.dat"
simulationPeriod = 500
startLocation = Location(0, 0)
if os.path.isfile(path):
    inp = open(path, "rb")
    dataDict = pickle.load(inp)
    inp.close()
    # Reading old game state
    simulationPeriod = dataDict["SimulationPeriod"]
    loc = dataDict["EndLocation"]
    location = Location(loc[0], loc[1])
    direction = dataDict["EndDirection"]
    setStatusText("Game state restored.")
else:
    location = startLocation
    direction = 0

clark = Lobster()
addActor(clark, startLocation)
clark.setLocation(location)
clark.setDirection(direction)
setSimulationPeriod(simulationPeriod)

while not isDisposed():
    delay(100)
gameData = {"SimulationPeriod": getSimulationPeriod(),
            "EndLocation": [clark.getX(), clark.getY()],
            "EndDirection": clark.getDirection()}
out = open(path, "wb")
pickle.dump(gameData, out)
out.close()
print "Game state saved"
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Il est possible de sauvegarder un dictionnaire vers un fichier binaire à l'aide de la méthode **pickle.dump().**Comme le fichier est au format binaire, il est impossible de l'ouvrir dans un éditeur et de le modifier.

EXERCICES

1. Rechercher les anagrammes dans le fichier *words-1$.txt* (deux mots possédant exactement les mêmes lettres mais dans un ordre différent, par exemple maison et aimons) en ignorant la casse. Écrire les anagrammes que vous avez trouvés dans un fichier *anagram.txt.*

2. Le texte suivant a été crypté par anagramme : les mots originaux ont été substitués par des mots dont les lettres sont permutées.

   IINFHS SOLOHC AEMK SIWREKOFR

   Décrypter ce texte à l'aide de la liste de mots (*words-1$.txt)* [**plus...**].

3. Créer votre propre texte crypté pouvant être décrypté de manière non ambigüe à l'aide de la liste de mots.

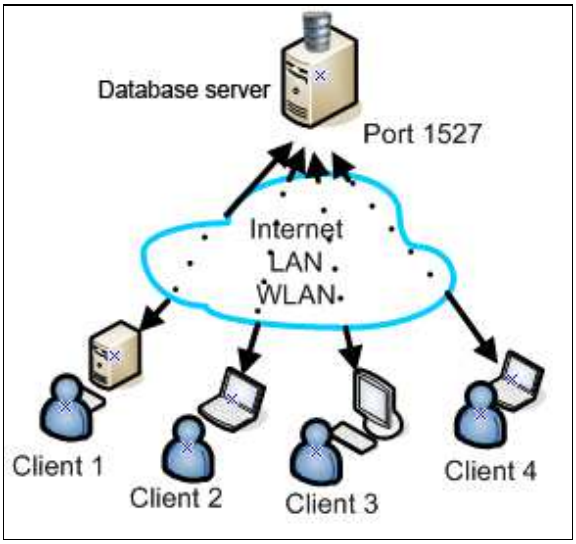
9.2 BASES DE DONNÉES EN LIGNE

INTRODUCTION

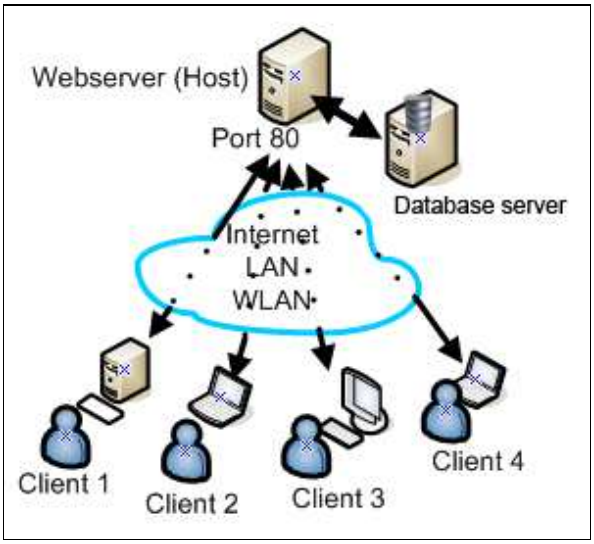
Les bases de données sont extrêmement importantes dans notre monde actuel. Leur rôle principal consiste à stocker des données de manière structurée pour qu'il soit facile d'y accéder à l'aide de critères de recherche permettant de les mettre en relation. En raison de la forte interconnexion du réseau Internet et de l'usage extrêmement répandu des réseaux sociaux, d'énormes montagnes de données sont actuellement stockées dans des millions de bases de données. Il est de ce fait essentiel de comprendre comment les données sont gérées au sein des bases de données. Cela vous permettra également de mieux apprécier les risques liés à l'utilisation de systèmes basés sur le stockage de données.

Dans la plupart des bases de données informatisées, les informations sont stockées sous forme de tables reliées entre elles, raison pour laquelle on parle de **bases de données relationnelles**. Afin de gérer efficacement toutes ces tables, il est nécessaire de disposer d'un puissant logiciel appelé système de gestion de base de données relationnelle (SGBDR), ce qui se traduit en anglais par *relational database management system* (RDBMS). Il faut donc concevoir les bases de données comme l'ensemble des données qu'elles contiennent et les outils complexes permettant de gérer ces dernières.

Les bases de données simples peuvent être hébergées localement sur un ordinateur personnel et être gérées par une seule personne. Dans cette catégorie de bases de données se trouvent par exemple les listes de CDs ou de livres. Mais la plupart des bases de données sont hébergées sur l'Internet et sont de ce fait accessibles simultanément par plusieurs utilisateurs (bases de données en ligne). Ces systèmes client-serveur comportent un ordinateur qui joue le rôle de serveur de base de données (également appelé hôte) et de nombreux clients distribués. L'échange de données avec le serveur de base de données se fait par une connexion TCP, exactement comme pour un serveur Web, à la différence que le port IP est différent (3306 pour MySQL ou 1527 pour Derby). Une application spécifique développée dans un langage de haut niveau tel que Python s'exécute sur le client et se connecte à l'hôte distant.



Connexion directe



Connexion indirecte

La plupart du temps, le client n'est pas connecté directement au serveur de base de données mais plutôt au travers d'un serveur Web. Dans ce cas, l'ordinateur client ne fait qu'exécuter une application Web au sein de son navigateur Web et le programme en interaction avec la base de données s'exécute sur le serveur Web qui doit également être développé dans un langage de programmation approprié, très souvent le PHP. Que l'on développe un programme en connexion

directe ou indirecte avec le serveur de base de données, il est nécessaire de disposer de solides connaissances de programmation en lien avec les bases de données. [**plus...**].

CONCEPTS DE PROGRAMMATION : *Base de données, table, serveur de base de données, gestion d'exceptions avec try-except*

VOTRE PROPRE SERVEUR DE BASE DE DONNÉES

Il est délicat d'accéder directement à un serveur de base de données présent sur Internet en raison des fortes restrictions sécuritaires visant à éviter que n'importe qui aille stocker ou modifier n'importe quelle donnée n'importe où. Pour cette raison, vous allez installer votre propre serveur de base de données sur votre PC pour que vous puissiez l'utiliser directement en local ou depuis d'autres postes connectés au même réseau local LAN ou WLAN. Il est généralement nécessaire de s'authentifier auprès d'un serveur de base de données à l'aide d'un nom d'utilisateur et d'un mot de passe [**plus...**].

Dans l'exemple suivant, vous allez installer Derby, un produit libre de la fondation Apache qui a également développé le très fameux serveur Web Apache. Vous pourriez très bien aussi utiliser une autre base de données telle que MySQL. Pour installer Derby, il faut suivre les étapes décrites ci-dessous:

1. Télécharger le fichier *tjderby.zip* depuis ce **lien.**
2. Décompacter cette archive et copier les fichiers qu'elle contient dans le sous-dossier *Lib du dossier dans lequel se trouve l'archive tigerjython2.jar*.
3. Se rendre dans le dossier *Lib* à l'aide d'un terminal en ligne de commandes
   et démarrer le serveur à l'aide de la commande

   ```
   java -jar derbynet.jar start
   ```

   Attention! Il est nécessaire pour cela de disposer des privilèges administrateurs sur la machine [**plus...**].

Comme alternative vous pouvez démarer le serverv par *les fichiers startderby.bat* (pour Windows) ou *startderby* (pour Linux/Mac, donner les droits en execution) que vous trouvez dans le dossier *Lib.* Il est pratique de créer un lien vers ces fichiers et de le placer sur votre bureau pour pouvoir lancer le serveur en un clic. S'il est démarré de cette manière, le serveur ne sera accessible qu'en local (localhost). Pour le rendre accessible depuis le réseau local, il est nécessaire de démarrer le serveur en spécifiant l'adresse IP de l'interface réseau connectée au LAN. Ainsi, si votre adresse IP est 10.1.1.123, il faudra exécuter la commande

```
java -jar derbynet.jar start -h 10.1.1.123
```

ou, en utilisant le script de démarrage,
starderby 10.1.1.123.

Le serveur de base de données peut gérer plusieurs bases de données simultanément. Il est possible d'accéder à une base de données en connaissant son nom ainsi qu'une combinaison valide de nom d'utilisateur et mot de passe spécifique. Pour se connecter et utiliser une base de données depuis un programme Python, il faut utiliser la fonction *getDerbyConnection(). Si la base de données en question n'est pas encore existante, elle sera automatiquement créée.*

Puisque nous allons développer un système de réservation de billets pour la salle de concert imaginaire Casino, nommez votre base de données *casino* et utilisez *admin/solar* comme nom d'utilisateur / mot de passe.

Une fois la connexion établie avec succès, la fonction *getDerbyConnection() retourne un objet connexioncon dont la méthode cursor() permet d'obtenir une clé d'accès (curseur) à la base de données. Une fois toutes ces étapes réalisées avec succès, toutes les portes de la base de données vous sont ouvertes depuis votre programme Python. On peut considérer une base de données comme un objet auquel on peut envoyer des commandes qui vont être scrupuleusement exécutées. Les commandes sont écrites dans un langage plus ou moins standardisé nommé SQL*

(Structured Query Language) qui est fortement basé sur une sorte d'argot anglais, de sorte qu'il est compréhensible par pratiquement n'importe qui. Il suffit, pour envoyer une commande SQL, de l'écrire dans une chaine de caractères et de l'envoyer au serveur grâce à la méthode *execute(SQL)*. Pour rendre les instructions plus lisibles, on a l'habitude d'écrire les éléments propres au langage SQL en lettres capitales. Il est cependant tout à fait possible (mais pas conseillé) d'utiliser des minuscules ou un mélange des deux.

Comme première étape, il est nécessaire de créer une table à l'aide de l'instruction SQL *CREATE TABLE*. Comme d'habitude, les tables sont formées de lignes et de colonnes. Les colonnes, également appelées « champs », spécifient la nature des informations stockées dans la table. Il est nécessaire à ce titre de spécifier un nom et un type de données pour chacun des champs. Les types les plus courants sont listés dans le tableau suivant :

SQL data type (for Derby)	**Description**
VARCHAR	Chaines de caractères de longueur variable (<=255)
CHAR(M)	Chaines de caractères de longueur M <= 255
INTEGER	Nombre entier codé sur 4 octets
FLOAT	Nombre flottant codé sur 64 bits (précision double)
DATE	Date (java.sql.date)

Afin d'administrer le système de réservation de sièges du casino pour un concert donné, il faut spécifier une date au format yyyymmdd (y: année, m: mois, d: jour) à la table *res*. Nous ajouterons encore les champs *seat* pour le numéro du siège, *booked* qui contiendra les caractères "N" ou "Y" pour déterminer si le siège en question a été réservé, et un numéro de clients *cust* qui identifie de manière unique la personne ayant effectué la réservation.

```
#Db2a.py

from dbapi import *

username = "admin"
password = "solar"
dbname = "casino"
serverURL = "localhost"
#serverURL = "10.1.1.123"

con = getDerbyConnection(serverURL, dbname, username, password)
cursor = con.cursor()
SQL = "CREATE TABLE res_20140115 (seat INTEGER,booked CHAR(1),cust INTEGER)"
cursor.execute(SQL)
con.commit()
cursor.close()
con.close()
print "Table created"
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Si le programme s'exécute sans encombre, c'est que l'installation du serveur de base de données s'est déroulée avec succès. Si, au contraire, vous obtenez des messages d'erreur, reprenez le chapitre en relisant toutes les instructions une à une.

Avec la plupart des SGBDR, il est très courant que les commandes SQL ne soient effectivement prises en compte qu'après un appel à la méthode *commit()*. Cela permet de garantir, pour des opérations impliquant plusieurs requêtes (on parle de **transaction** dans le langage des bases de données), qu'elles soient exécutées en intégralité ou, qu'en cas d'erreur, elles ne soient pas

exécutées du tout. Finalement, il est nécessaire de libérer avec la méthode *close()* toutes les ressources utilisées pour la communication avec la base de données. Mémorisez attentivement la séquence de « nettoyage »

```
con.commit()
cursor.close()
con.close()
```

qu'il ne faudra jamais oublier par la suite.

Ce programme va générer un message d'erreur et s'interrompre si la table existe déjà, ce qui est très ennuyeux puisque toute tentative ultérieure de TigerJython pour se connecter à la base de données se soldera par un échec, à moins qu'il soit complètement redémarré.

GÉRER LES ERREURS AVEC TRY-EXCEPT

Si la table que le programme tente de créer existe déjà, l'appel à *execute()* va faire planter le programme. On dit en langage technique que le programme « lève » (*raise*) ou « lance » (*throw*) une exception, selon les terminologies. Les instructions subséquentes du programme ne seront alors pas exécutées, en particulier les instructions responsables de faire le ménage. En conséquence, certaines ressources nécessaires pour la connexion à la base de données ne sont pas libérées, ce qui peut bloquer d'autres programmes et même tout le système. Il est de ce fait très important d'attraper au vol de telles erreurs.

Gérer une exception levée par Python est très simple (on pourrait aussi dire que l'on « attrape » une exception « lancée » par Python). On place les instructions qui sont susceptibles de lever une exception dans un bloc *try-except*. Si l'exception en question survient, le programme passe alors immédiatement dans le bloc *except*, après quoi l'exécution du programme se poursuit avec l'instruction qui suit directement le bloc *try-except*. De manière facultative, on peut également rajouter un bloc *else* qui sera exécuté en l'absence de toute exception.

Par la suite, nous placerons donc toutes nos commandes SQL susceptibles de lever une exception dans un bloc *try-except* de manière à gérer toutes les éventuelles exceptions qu'elles pourraient engendrer.

```
from dbapi import *

username = "admin"
password = "solar"
dbname = "casino"
serverURL = "localhost"
#serverURL = "10.1.1.123"

con = getDerbyConnection(serverURL, dbname, username, password)
cursor = con.cursor()
try:
   SQL = "CREATE TABLE res_20140115 (seat INTEGER,booked CHAR(1),cust INTEGER)"
   cursor.execute(SQL)
except Exception, e:
   print "SQL executing failed.", e
else:
   print "Table created"
con.commit()
cursor.close()
con.close()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Toutes les parties critiques d'un programme devraient être placées dans un bloc *try-except* de telle manière que le code de nettoyage puisse être exécuté même lorsque des exceptions sont levées.

La commande *except* peut obtenir par le biais du paramètre *e* d'importantes informations au sujet de l'erreur qui est survenue.

INSÉRER DES DONNÉES DANS UNE TABLE

Dans un second temps, nous allons insérer des données d'initialisation dans la table, notamment pour marquer tous les sièges comme disponibles et initialiser le numéro du client à 0. Pour ce faire, on utilise la commande INSERT du langage SQL. Pour le siège numéro 1, cela donnerait par exemple la requête

```
INSERT INTO res_20140115 VALUES (1, 'N', 0)
```

Cette requête ajoute une ligne dans notre table. Une ligne est également couramment désignée par l'expression **jeu de données** (*dataset* en anglais) ou enregistrement (*record* en anglais).

```
#Db2c.py

from dbapi import *

table = "res_20140115"
username = "admin"
password = "solar"
dbname = "casino"
serverURL = "localhost"
#serverURL = "10.1.1.123"
con = getDerbyConnection(serverURL, dbname, username, password)

cursor = con.cursor()
try:
    for seatNb in range(1, 31):
        SQL = "INSERT INTO " + table + " VALUES (" + str(seatNb) + ",'N',0)"
        cursor.execute(SQL)
except Exception, e:
    print "SQL executing failed. ", e
else:
    print "Table initialized"
con.commit()
cursor.close()
con.close()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Si le programme est exécuté deux fois, l'enregistrement sera également inséré deux fois dans la base de données.

LIRE DES DONNÉES DEPUIS UNE TABLE

Vous devez à présent vous demander si les données se trouvent réellement dans la table. Dans tous les cas, les SGBDR professionnels actuels sont tellement bétonnés que l'on peut raisonnablement supposer, en l'absence de tout message d'erreur, que la transaction avec la

base de données s'est déroulée comme prévu. Pour accéder aux données présentes dans la table, on utilise l'instruction SQL la plus connue :

```
SELECT * FROM res_20140115
```

L'astérisque (caractère de remplacement) permet de lire l'ensemble des enregistrements de la table. Suite à un appel à la méthode *execute()*, il est possible d'obtenir les enregistrements retournés par le SGBDR grâce à la méthode *fetchall()* de l'objet *cursor*. Cette méthode retourne une liste de tuples (sorte de liste non mutable) représentant les enregistrements répondant à la requête. L'accès aux différents champs d'un enregistrement se fait à l'aide d'indices.

Il est important de pouvoir connaître le nombre d'enregistrements retournés par la commande *SELECT*. Ce nombre se trouve toujours enregistré dans la variable globale *rowcount* après une opération *SELECT*.

```
#Db2d.py

from dbapi import *

table = "res_20140115"
username = "admin"
password = "solar"
dbname = "casino"
serverURL = "localhost"
#serverURL = "10.1.1.123"

con = getDerbyConnection(serverURL, dbname, username, password)
cursor = con.cursor()
try:
    SQL = "SELECT * FROM " + table
    cursor.execute(SQL)
except Exception, e:
    print "SQL execution failed.", e
else:
    nbRecord = cursor.rowcount
    print "Number of records:", nbRecord
    result = cursor.fetchall() # list of tuples
    for record in result:
        print "seatNb:", record[0], " booked:", record[1]," cust:", record[2]
con.commit()
cursor.close()
con.close()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

En plus de *fetchall(), il est également possible d'utiliserfetchmany(n)* et *fetchone()* pour récupérer les données du curseur. À chaque appel de l'une de ces méthodes, le curseur est pour ainsi dire avancé comme un pointeur du nombre d'enregistrements lus (d'où le nom *curseur*). Un appel subséquent de l'une des méthodes *fetchall()*, *fetchmany()* ou *fetchone() lira donc les enregistrements à partir de la dernière position du curseur. Lorsque le curseur atteint la fin de la table, ces méthodes retournent la valeur None.*

SUPPRIMER UNE TABLE

Pour terminer cette petite interaction avec la base de données, supprimons la table que nous venons de créer. Dans le jargon des bases de données, on dit qu'on « dépose la table » (*drop* en anglais). La commande SQL correspondante est :

```
DROP TABLE res_20140115
```

Après cette opération, la base de données *casino ne comporte plus aucune table définie par l'utilisateur*.

```
from dbapi import *

table = "res_20140115"
username = "admin"
password = "solar"
dbname = "casino"
serverURL = "localhost"
#serverURL = "10.1.1.123"

con = getDerbyConnection(serverURL, dbname, username, password)
cursor = con.cursor()
try:
    SQL = "DROP TABLE " + table
except Exception, e:
    print "SQL executing failed. ", e
else:
    print "Table removed"
con.commit()
cursor.close()
con.close()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Après la suppression de la table, les programmes lisant des données depuis la table en question lèveront une exception avec un message d'erreur. Il sera encore nécessaire de les adapter de manière appropriée pour gérer cette situation particulière.

La base de données nommée *pythondb continue d'exister même après la suppression de la table*. De plus, le sous dossiers *pythondb du dossier dans lequel vous avez installé le serveur Derby (en l'occurrence Lib/pythondb) contient encore de nombreux fichiers. Pour effacer toute tracede cet exercice, on peut simplement supprimer le dossier pythondb*.

EXERCICES

1. Créer un nouveau système de réservation de places permettant d'effectuer et d'annuler la réservation d'un siège. Pour ce faire, n'oubliez pas de démarrer le serveur *Derby* et exécutez encore les programmes Db2a.py et Db2c.py. Ceux-ci vont créer une nouvelle table et ajouter 30 enregistrements correspondant à des sièges non réservés.

   Le programme suivant, qui est une simplification de Db2d.py, permet d'afficher l'ensemble des enregistrements de la table.

   ```
   from dbapi import *

   username = "admin"
   password = "solar"
   dbname = "casino"
   serverURL = "localhost"
   table = "res_20140915"

   def showAll():
       SQL = "SELECT * FROM " + table
       cursor.execute(SQL)
       con.commit()
       result = cursor.fetchall()
   ```

```
        for record in result:
            print "seatNb:", record[0]," booked:",record[1],"customer:",record[2]

con = getDerbyConnection(serverURL, dbname, username, password)
cursor = con.cursor()
showAll()
cursor.close()
con.close()
```

2. Pour réserver le siège 6 pour le client 33, il faut ajouter une instruction *UPDATE* au programme ci-dessus avant l'appel à *showAll()*.

```
SQL = "UPDATE " + table + " SET booked='Y', cust=33 WHERE seat=6"
cursor.execute(SQL)
```

   Réserver également le siège numéro 5 pour le même client. De plus, réservez les sièges 10, 11 et 12 pour le client numéro 34 et les sièges 17 et 18 pour le client numéro 35.

3. Il est possible d'afficher l'ensemble des sièges réservés à l'aide de l'instruction

```
SQL = "SELECT * FROM " + table + " WHERE booked='Y'"
cursor.execute(SQL)
```

   a. Modifier la fonction *showAll()* de telle manière que seuls les sièges réservés soient affichés
   b. Modifier la requête pour n'afficher que les sièges réservés par le client numéro 34.

4. Le client 34 a annulé toutes ses réservations. Implémenter la requête de mise à jour correspondante.

5. Suite à une rénovation de la salle de concert, les sièges 24 – 30 ne sont plus disponibles. Supprimer ces enregistrements de la table.

   Pour ce faire, utiliser une instruction DELETE selon le schéma suivant

```
SQL = "DELETE * FROM " + table + " WHERE ..."
cursor.execute(SQL)
```

MATÉRIEL SUPPLÉMENTAIRE

PROGRAMMES DE GESTION DE BASE DE DONNÉES

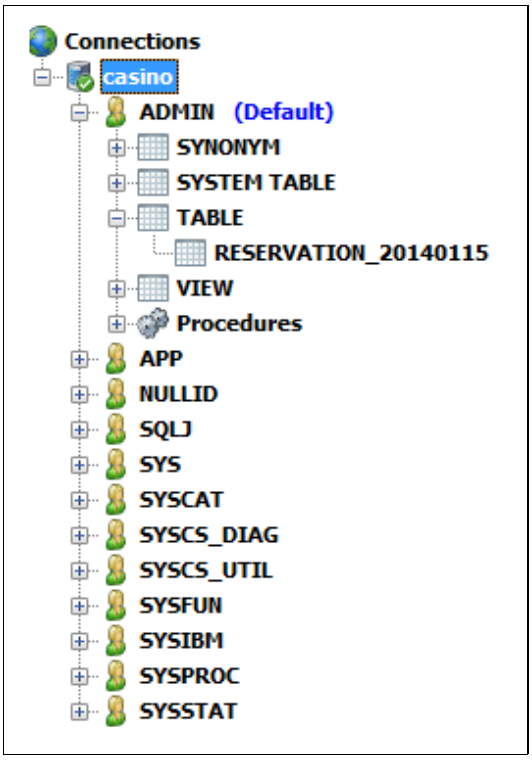
En tant qu'administrateur de base de données, il est plus pratique de pouvoir gérer ses bases de données sans avoir à écrire des programmes Python. Il existe à cette fin de nombreux outils administratifs disponibles en téléchargement sur le Web. Ceux-ci ne sont cependant pas toujours libres. On distingue typiquement les outils en ligne de commande et les outils disposant d'une interface graphique. Alors que les professionnels sont généralement très à l'aise dans une interface en lignes de commandes, les utilisateurs occasionnels préféreront les programmes disposant d'une interface graphique.

DBVisualizer est un outil bien connu disposant d'une telle interface graphique et il est disponible en téléchargement gratuitement depuis *le lien noté ci-dessous [**plus...**]. On pourra par contre l'utiliser facilement pour exécuter des requêtes SQL et observer directement leurs effets sur les données. Voici comment l'installer :*

1. Télécharger le programme d'installation correspondant à votre plateforme **http://www.dbvis.com/download**
2. Exécuter le fichier téléchargé et choisir les options par défaut.

3. S'assurer que le serveur Derby soit démarré, créer la base de données *casino* ainsi qu'une table *res_20140115 et initialiser les enregistrements*
4. Démarrer *DBVisualizer* et sélectionner *Tools* | *Connection Wizard*. Dans la boîte de dialogue, saisir n'importe quel alias de connexion, par exemple *casino*. Dans la boîte de dialogue *Select Database Driver*, sélectionner *JavaDB/Derby Server*.
5. Dans la boîte de dialogue suivante, fournir le nom d'utilisateur *admin*, le mot de passe *solar* et le nom de la base de données *casino*. Cliquer ensuite sur *casino*.

   Vous devriez voir à présent un écran semblable à l'image ci-dessous dans la fenêtre de navigation. Il est possible d'ouvrir une fenêtre en double-cliquant sur la table RES_2014015. Pour afficher le contenu la table, il suffit de cliquer sur l'onglet *Data*.



9.3 SYSTÈME DE RÉSERVATION EN LIGNE

INTRODUCTION

De nos jours, les réservations de places de spectacle sont effectuées presque exclusivement au moyen de systèmes de réservation en ligne qui utilisent des bases de données pour gérer les données. Des spécialistes en bases de données pourvus de très grandes compétences en programmation sont responsables du développement de ce genre de systèmes.

CONCEPTS DE PROGRAMMATION : Systèmes multi-utilisateurs, *conflits d'accès, erreur sporadique*

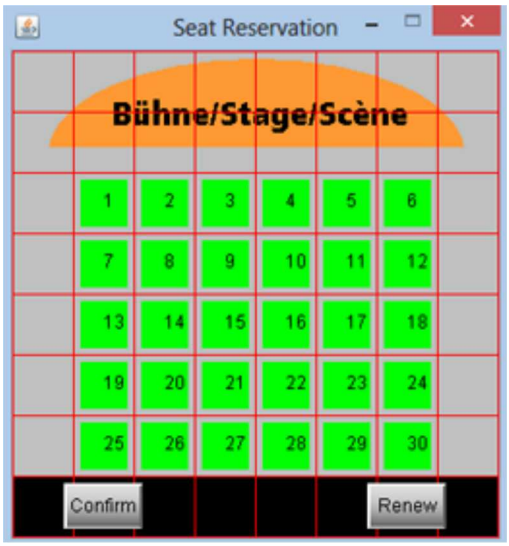
SYSTÈME DE RÉSERVATION, ÉTAPE 1 : INTERFACE GRAPHIQUE

Il faut prévoir une interface graphique attractive pour présenter le système à l'utilisateur. Une telle interface peut être réalisée au travers d'une application Desktop classique ou d'une application Web. L'interface d'un système de réservation affiche généralement la disposition des sièges dans la salle de spectacle ou dans l'avion pour permettre à l'utilisateur de réserver son siège de manière fondée, sur la base d'une bonne représentation spatiale. La disponibilité des sièges est rendue immédiatement apparente par un code couleur : rouge pour les sièges réservés et vert pour ceux qui sont encore libres.

L'utilisateur peut se contenter de cliquer sur un siège vide pour en demander la réservation, de sorte qu'il va changer de couleur. Cette sélection n'est à ce stade pas encore communiquée à la base de données puisque l'utilisateur va généralement réserver plusieurs chaises ou changer d'avis au cours de sa réservation. Le processus de réservation définitive n'est initié que lorsque l'utilisateur confirme sa réservation en cliquant sur un bouton approprié.

La bibliothèque de jeux *JGameGrid* est idéale pour implémenter cette interface graphique puisque les sièges sont généralement arrangés selon une structure de grille. L'identifiant de sprite (0, 1, 2) peut être utilisé pour permettre de changer l'état de disponibilité des sièges (libre, en cours de réservation, réservé).

Dans le cas présent, on modélise une petite salle de spectacle ne disposant que de 30 sièges numérotés de 1 à 30 et disposés dans la grille représentée ci-contre.



Les deux boutons sont réalisés grâce à la classe *GGButton*. Pour pouvoir détecter les clics sur les boutons, il est nécessaire de définir une nouvelle classe de boutons dérivant de *GGButtons* et de *GGButtonListener (héritage multiple)*. La méthode *buttonPressed()* est invoquée lors d'un clic sur l'un des deux bouton. On peut déterminer lequel des boutons a été cliqué sur la base du paramètre *button*.

Le programme suivant constitue déjà une interface graphique pleinement fonctionnelle. Il lui manque cependant encore la connexion à la base de données [**plus...**] .

```
from gamegrid import *

def toLoc(seat):
    i = ((seat - 1) % 6) + 1
    k = ((seat - 1) // 6) + 2
    return Location(i, k)

def toSeatNb(loc):
    if loc.x < 1 or loc.x > 6 or loc.y < 2 or loc.y > 6:
       return None
    i = loc.x - 1
    k = loc.y - 2
    seatNb = k * 6 + i + 1
    return seatNb

class MyButton(GGButton, GGButtonListener):
    def __init__(self, imagePath):
        GGButton.__init__(self, imagePath)
        self.addButtonListener(self)

    def buttonClicked(self, button):
        if button == confirmBtn:
            confirm()
        if button == renewBtn:
            renew()

def renew():
    setStatusText("View refreshed")

def confirm():
    for seatNb in range(1, 31):
       if seats[seatNb - 1].getIdVisible() ==  1:
           seats[seatNb - 1].show(2)
           refresh()
    setStatusText("Reservation successful")

def pressCallback(e):
    loc = toLocation(e.getX(), e.getY())
    seatNb = toSeatNb(loc)
    if seatNb == None:
        return
    seatActor = seats[seatNb - 1]
    if seatActor.getIdVisible() == 0:  # free
        seatActor.show(1) # option
        refresh()
    elif seatActor.getIdVisible() == 1:  # option
       seatActor.show(0) # free
       refresh()


makeGameGrid(8, 8, 40, None, "sprites/stage.gif", False,
             mousePressed = pressCallback)
addStatusBar(30)
setTitle("Seat Reservation")
setStatusText("Please select free seats and press 'Confirm'")
confirmBtn = MyButton("sprites/btn_confirm.gif")
renewBtn = MyButton("sprites/btn_renew.gif")
addActor(confirmBtn, Location(1, 7))
addActor(renewBtn, Location(6, 7))
seats = []
for seatNb in range(1, 31):
    seatLoc = toLoc(seatNb)
    seatActor = Actor("sprites/seat.gif", 3)
    seats.append(seatActor)
    addActor(seatActor, seatLoc)
    addActor(TextActor(str(seatNb)), seatLoc)
show()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

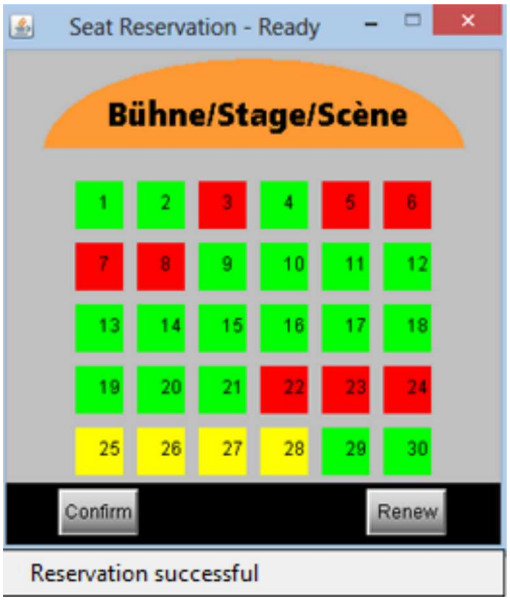
MEMENTO

Le passage d'un numéro de siège en sa position dans la grille (Location) et vis-versa est réalisé au sein de deux fonctions de transformation. Le nom de telles fonctions de conversion est généralement préfixé de *to* (vers en anglais).

SYSTÈME DE RÉSERVATION, ÉTAPE 2 : CONNEXION À LA BASE DE DONNÉES

Les bases de données en ligne sont des systèmes multi-utilisateurs. Du fait que de nombreux clients manipulent les données en même temps, de nombreux problèmes de conflits d'accès peuvent survenir dans certaines situations. Les problèmes de ce type sont par nature sporadiques et donc très difficilement gérables. Le scénario suivant décrit un cas typique de conflit d'accès pour notre système de réservation : Deux clients A et B effectuent une réservation en même temps. Au début, les deux clients A et B vont interroger la base de données pratiquement en même temps pour connaître les sièges encore libres. Les deux reçoivent donc exactement la même information et possèdent donc la même vision de la situation (vert = siège libre, rouge = siège réservé).

Aussi bien A que B peut à présent, par un clic de souris, réserver temporairement des sièges qui seront alors affichés en jaunes (réservation temporaire). Ces réservations temporaires ne sont à ce stade pas encore communiquées à la base de données centrale puisque le client pourrait encore changer d'avis ou réserver davantage de sièges. Assez rapidement, le client A, plus rapide à se décider, confirme sa réservation, ce qui a pour effet de réserver les sièges définitivement dans la base de données centrale en passant les sièges à l'état réservé *booked='Y'*). Moins rapide à la détente, l'utilisateur B n'a pas connaissance de ces modifications du côté serveur puisque la base de données ne peut pas directement lui donner un feedback de manière non sollicitée. De plus, le client B n'est pas non plus en contact direct avec le client A. [**plus...** Juste un instant plus tard, le client B valide également sa réservation en cliquant sur le bouton de confirmation.



Comme le stipule la loi de Murphy « S'il y a un truc qui peut foirer, ça va foirer », il se trouve que les clients A et B ont réservé exactement les mêmes sièges. Que va-t-il se produire ? Les sièges vont-ils alors être attribués à l'utilisateur B qui est le dernier à écrire sa réservation vers la base de données ou sont programmes va-t-il se planter ?

Il existe une solution toute simple à ce problème : juste avant que la réservation temporaire ne soit confirmée, il faut interroger à nouveau la base de données pour s'assurer que les sièges demandés n'aient pas été réservés par un autre client dans l'intervalle. Si tel est malheureusement le cas, la base de données n'a pas d'autre choix que d'avertir poliment l'utilisateur B qu'il a manqué de rapidité et que les sièges qu'il demande ont déjà été attribués [**plus..**]

Remarque : L'exécution du programme suivant présuppose que le serveur de bases de données est démarré et que la table *res_20140115* a été créée et remplie avec les données d'initialisation (voir chapitre précédent).

```
from dbapi import *
from gamegrid import *

table = "res_20140115"
username = "admin"
password = "solar"
```

```
dbname = "casino"
serverURL = "localhost"
#serverURL = "10.1.1.123"

def toLoc(seat):
    i = ((seat - 1) % 6) + 1
    k = ((seat - 1) // 6) + 2
    return Location(i, k)

def toSeatNb(loc):
    if loc.x < 1 or loc.x > 6 or loc.y < 2 or loc.y > 6:
       return None
    i = loc.x - 1
    k = loc.y - 2
    seatNb = k * 6 + i + 1
    return seatNb

def pressCallback(e):
    if not isReady:
        return
    loc = toLocation(e.getX(), e.getY())
    seatNb = toSeatNb(loc)
    if seatNb == None:
        return
    seatActor = seats[seatNb - 1]
    if seatActor.getIdVisible() == 0:  # free
        seatActor.show(1) # option
        refresh()
    elif seatActor.getIdVisible() == 1:  # option
       seatActor.show(0) # free
       refresh()

class MyButton(GGButton, GGButtonListener):
    def __init__(self, imagePath):
        GGButton.__init__(self, imagePath)
        self.addButtonListener(self)

    def buttonClicked(self, button):
        if not isReady:
            return
        if button == confirmBtn:
            confirm()
        if button == renewBtn:
            renew()

def renew():
    global isReady
    try:
        SQL = "SELECT * FROM " + table
        cursor.execute(SQL)
        con.commit()
    except:
        setStatusText("Fatal error. Restart and try again.")
        isReady = False
        return

    result = cursor.fetchall()
    for record in result:
        seatNb = record[0]
        isBooked = (record[1] != 'N')
        if isBooked:
            seats[seatNb - 1].show(2)
        else:
            seats[seatNb - 1].show(0)
    refresh()
    setStatusText("View refreshed")

def confirm():
    global isReady
    try:
```

```
        # check if seats is still available
        for seatNb in range(1, 31):
            if seats[seatNb - 1].getIdVisible() == 1:
                SQL = "SELECT * FROM "  + table + "  WHERE seat=" + str(seatNb)
                cursor.execute(SQL)
                result = cursor.fetchall()
                for record in result:
                    if record[1] == 'Y':
                        setStatusText("One of the seats are already taken.")
                        return
        isReserved = False
        for seatNb in range(1, 31):
            if seats[seatNb - 1].getIdVisible() == 1:
                SQL = "UPDATE "  + table + " SET booked='Y' WHERE seat=" + \
                      str(seatNb)
                cursor.execute(SQL)
                isReserved = True
            con.commit()
        renew()
        if isReserved:
            setStatusText("Reservation successful")
        else:
            setStatusText("Nothing to do")
    except Exception, e:
        setStatusText("Fatal error. Restart and try again.")
        isReady = False

isReady = False
makeGameGrid(8, 8, 40, None, "sprites/stage.gif", False,
             mousePressed = pressCallback)
addStatusBar(30)
setTitle("Seat Reservation - Loading...")
confirmBtn = MyButton("sprites/btn_confirm.gif")
renewBtn = MyButton("sprites/btn_renew.gif")
addActor(confirmBtn, Location(1, 7))
addActor(renewBtn, Location(6, 7))
seats = []
for seatNb in range(1, 31):
    seatLoc = toLoc(seatNb)
    seatActor = Actor("sprites/seat.gif", 3)
    seats.append(seatActor)
    addActor(seatActor, seatLoc)
    addActor(TextActor(str(seatNb)), seatLoc)
show()

con = getDerbyConnection(serverURL, dbname, username, password)
if con == None:
    setStatusText("Fatal error. Connection to database failed")
else:
    cursor = con.cursor()
    renew()

    setTitle("Seat Reservation - Ready")
    setStatusText("Select free seats and press 'Confirm'")
    isReady = True
    while not isDisposed():
        delay(100)
    cursor.close()
    con.close()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Dans cet exemple, on voit que l'on ne peut pas se contenter d'écrire un programme de telle manière qu'il se comporte correctement dans des conditions idéales, à savoir lorsqu'il est utilisé correctement. Au contraire, le programme doit être capable de se comporter de manière appropriée même lorsque les conditions sont moins favorables ou en cas d'erreur de

manipulation. Une interface graphique bien développée devrait toujours tenir l'utilisateur au courant de l'évolution de l'état du système pour qu'il puisse savoir si ses actions ont abouti. D'autre part, des manipulations interdites devraient être désactivées dans l'interface graphique.

Une façon courante de désactiver des actions au sein de l'interface graphique consiste à utiliser un fanion (flag) *isReady* qui déterminer si une saisie au clavier ou à la souris est permise ou non. Le fanion *isReady* est initialement réglé sur *False* jusqu'à ce que la connexion à la base de données soit établie, ce qui peut prendre quelques secondes pour une base de données distante. Les dysfonctionnements sont interceptés au sein d'un bloc *try-except* dont le bloc *except* mettra le fanion *isReady* à *False*.

Vous pouvez tester le système de réservation avec plusieurs fenêtres en démarrant successivement plusieurs fois TigerJython et en exécutant le programme client dans chacune d'elles.

EXERCICES

1. Étendre le système de réservation de telle manière qu'un clic sur le bouton de confirmation entraîne l'ouverture d'une boîte de dialogue demandant au client de saisir son numéro de client. Enregistrer le numéro de client récupéré sous forme de nombre entier dans la base de données. Utiliser l'appel *inputInt(prompt, False)* pour afficher la boîte de dialogue et lire la valeur puisque le deuxième paramètre *False* permet de ne pas fermer le programme lorsque l'utilisateur clique sur *Annuler* mais de plutôt retourner la valeur *None*.

2. Développer un outil administrateur affichant la disposition actuelle des sièges ainsi qu'une liste des sièges réservés avec le nom de client. Il devrait de plus être possible d'annuler une réservation en cliquant sur un siège rouge. Pour afficher la liste des réservations, ouvrir une fenêtre de console avec *console = console.init().* La méthode *console.println(text)* permet d'écrire dans la console ligne à ligne. *Pour utiliser la console, il faut effectuer l'importation suivante en début du programme : from ch.aplu.util import Console*

3. Développer un outil permettant de créer une table *client* dont les champs sont le numéro de client, le nom et le prénom du client. Modifier l'outil d'administration de l'exercice 2 de telle sorte que le nom des clients apparaissent également dans la liste des sièges réservés.

Requêtes SQL

SELECT * [column] **FROM** table [**WHERE** condition] [**ORDER BY** column [asc|desc]]

Les options indiquées entre crochets carrés sont facultatives. Voici quelques exemples de requêtes SELECT:

SELECT * FROM tab	Retourne tous les enregistrements de la table *tab*
SELECT name, firstname FROM tab	Idem mais en demandant uniquement les champs *name* et *firstname*
SELECT * FROM tab ORDER BY name	Tous les enregistrements de la table *tab* triés de manière ascendante selon *name*
SELECT * FROM tab WHERE salutation = 'Mr.' ORDER BY name	Tous les enregistrements pour lesquels le champ *salutation* vaut “Mr.”
SELECT * FROM tab WHERE name = 'Meier' and firstname = 'Luka'	Retourne les enregistrements qui satisfont à la fois aux condition *name = 'Meier'* et *firstname = 'Luka'*.
SELECT * FROM tab WHERE name = 'Meier' or name = 'Mayer'	Retourne les enregistrements satisfaisant à l’une ou l’autre des conditions
SELECT * FROM tab WHERE name in ('Meier', 'Meyer', 'Muller')	Retourne les enregistrements dont le champ *name* figure parmi les alternatives mentionnées entre parenthèses
SELECT * FROM tab WHERE name LIKE '%house% '	Retourne tous les enregistrements tels que la chaine de caractère "house" est contenue dans leur champ *name*
SELECT * FROM tab WHERE annee between 1999 and 2014	Les nombres peuvent être spécifiés sans les guillemets
SELECT count (*) FROM tab	Déterminer le nombre d’enregistrements dans la table
SELECT concat (firstame, ' ', name) as fname FROM tab	Retourne tous les enregistrements de la table *tab* en ajoutant dynamiquement un nouveau champ *fname* résultant de la concaténation des champs *name* et *firstname*
SELECT sum(price) FROM tab	Détermine la somme des montants de l’ensembles des réservations de la table

UPDATE table **SET** column1 = value1, [column2 = value2], [...] [**WHERE**condition]

UPDATE tab SET institut = 'EPFL'	Met à jour le champ *institut* de tous les enregistrements de la table *tab* en y stockant comme nouvelle valeur la chaîne 'EPFL'
UPDATE tab SET booked='Y', cust=33 WHERE seat=6	Met à jour les champs *booked* et *cust* de l’enregistrement pour lequel *seat=6*.
UPDATE tab SET solutation = 'Mme' WHERE salutation = 'f'	Mettre la valeur 'Ms' dans le champ *salutation* pour tous les enregistrements pour lesquels *salutation* = 'f'
UPDATE tab SET price = price * 1.52	Dans un UPDATE, on peut également déterminer par calcul la nouvelle valeur du champ sur la base de la valeur actuelle des champs.

DELETE FROM table **[WHERE** condition**]**

DELETE FROM tab	Supprime tous les enregistrements de la table *tab*
DELETE FROM tab WHERE name = "Meier"	Supprime tous les enregistrements de la table *tab* pour lesquels le champ *name* vaut *'Meier'*

chapitre dix

EFFICACITÉ & LIMITATIONS

Objectifs d'apprentissage

- Être capable d'illustrer par des exemples qu'il existe des problèmes pouvant être formulés de manière algorithmique mais non résolubles par ordinateur.
- Être capable d'expliquer clairement ce que l'on entend par complexité polynômiale et non polynômiale d'un programme.
- Être capable d'expliquer le problème d'arrêt en s'appuyant sur l'exemple de l'algorithme 3n+1.
- Être en mesure d'expliquer ce que l'on entend par « explosion combinatoire ».
- Être capable d'effectuer une recherche dans un graphe par retour-arrière (backtracking).
- Connaître quelques méthodes de chiffrement classiques et être capable de les implémenter dans un programme.
- Connaitre le concept de machine à états finis et être en mesure d'implémenter une machine à états finis.

10.1 COMPLEXITÉ DE TRI

INTRODUCTION

Les ordinateurs sont bien davantage que de simples machines à calculer. On sait par exemple qu'une proportion importante du temps processeur mondial est utilisé pour trier des données pour y faire des recherches. De ce fait, il est crucial pour un programme informatique de ne pas se contenter de livrer le résultat correct mais encore de le faire de manière optimisée. L'optimisation d'un programme touche principalement les aspects suivants :

- Sa longueur
- Sa structure et sa clarté (lisibilité)
- Le temps CPU nécessaire à son exécution
- Son utilisation de la mémoire de travail (RAM)

Fondamentalement, dans l'idéal, il faudrait toujours envisager l'optimisation du code comme faisant partie intégrante de la conception du programme.
Dans ce chapitre, nous allons examiner l'optimisation du temps d'exécution d'un algorithme de tri. Nous aborderons également les limites de l'informatique et des ordinateurs au travers d'un exemple qui, bien que possédant certainement une solution algorithmique, ne peut être résolu même à l'aide de l'ordinateur le plus puissant de la planète. On parle alors de **problème insoluble**.

CONCEPTS DE PROGRAMMATION: *Complexité, temps d'exécution, ordre d'un algorithme, méthodes de tri, redéfinition d'opérateurs.*

TRIER COMME DES ENFANTS

Trier et réordonner un ensemble d'objets à l'aide des opérateurs de comparaison *plus grand que, plus petit que* et *égal* [**plus...**] reste une tâche standard en informatique. Bien que tous les langages de programmation mettent à disposition des bibliothèques de fonctions permettant d'effectuer des tris, il est absolument indispensable d'inclure les concepts théoriques du tri dans votre bagage informatique car vous rencontrerez toujours des situations dans lesquelles il est nécessaire d'implémenter et optimiser son propre algorithme de tri spécifique.

On désigne par le terme « ensemble » une collection d'objets non triés. Les objets sont stockés dans une structure de données à une dimension. Les listes se prêtent particulièrement bien à cet égard.[**plus...**].

Le programme suivant illustre le tri à l'aide de petits lutins représentés à l'écran par des acteurs de la bibliothèque de jeu *JGameGrid*. On peut facilement afficher leur image de sprite dans une grille [**plus...**]. La hauteur de l'image de sprite (en pixels) sert également de mesure de la taille du personnage.

Les algorithmes sont souvent inspirés directement de processus que nous appliquons régulièrement dans notre vie de tous les jours. Si l'on demande à des enfants d'expliquer comment ils procèdent pour trier un ensemble d'objets d'après leur taille, ils décriront souvent le processus de la manière suivante : « *je prends le plus petit objet (ou le plus grand) et je le place tout à gauche* ». Cette façon de procéder semble tout-à-fait raisonnable mais présente un problème pour l'ordinateur car celui-ci n'est pas capable de déterminer le plus grand ou le plus petit objet en un coup d'œil. Afin de trouver le plus petit, il est obligé de parcourir un à un tout l'ensemble d'objets et de tous les comparer deux à deux. Pour être en mesure d'implémenter ce processus de tri, appelé en l'occurrence **children sort**, il faut une fonction *getSmallest(row)* qui retourne le plus petit lutin de la liste examinée. On peut procéder comme suit.

On sauvegarde le premier élément de la liste dans la variable *smallest* et on parcourt tous les éléments suivants dans une boucle *for*. Si l'on tombe sur un élément plus petit que celui temporairement enregistré dans la variable *smallest*, on remplace *smallest par celui-ci et on continue le parcours de la liste*.

Pour le *children sort*, on utilise deux listes : la première, *startList*, contient les objets donnés et l'autre liste, *targetList*, est initialement vide. On recherche le plus petit élément de *startList*, on l'en supprime et on le rajoute à la fin de la liste *targetList*. On répète ce processus jusqu'à ce que *startList soit vide*.

```
from gamegrid import *
import random

def bodyHeight(dwarf):
    return dwarf.getImage().getHeight()

def updateGrid():
   removeAllActors()
   for i in range(len(startList)):
       addActor(startList[i], Location(i, 0))
   for i in range(len(targetList)):
       addActor(targetList[i], Location(i, 1))

def getSmallest(li):
    global count
    smallest = li[0]
    for dwarf in li:
        count += 1
        if bodyHeight(dwarf) < bodyHeight(smallest):
            smallest = dwarf
    return smallest

n = 7

makeGameGrid(n, 2, 170, Color.red, False)
setBgColor(Color.white)
show()

startList = []
targetList = []

for i in range(0 , n):
    dwarf = Actor("sprites/dwarf" + str(i) + ".png")
    startList.append(dwarf)
random.shuffle(startList)
updateGrid()
setTitle("Children Sort. Press <SPACE> to sort...")
count = 0
while not isDisposed() and len(startList) > 0:
    c = getKeyCodeWait()
    if c == 32:
        smallest = getSmallest(startList)
        targetList.append(smallest)
        startList.remove(smallest)
        count += 1
```

```
            setTitle("Count: " + str(count) + " <SPACE> for next step...")
            updateGrid()
    setTitle("Count: " + str(count) + " All done")
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

L'algorithme du *children sort* nécessite, en plus de la liste d'objets à trier de longueur *n*, une seconde liste dont la longueur sera au bout du compte aussi de longueur *n*. Si la taille de la liste à trier *n* est très grande, cela peut poser un sérieux problème de mémoire [**plus...**].

On peut facilement déterminer le nombre d'opérations élémentaires nécessaires pour résoudre le problème : indépendamment de l'arrangement des objets dans la liste donnée, il faut forcément commencer par parcourir tous les *n* éléments de la liste puis, lors du deuxième passage, *n-1* éléments etc … De plus, il faut encore à chaque fois déplacer vers la liste destination le plus petit élément trouvé dans la liste à trier. Le nombre *c* d'opérations nécessaires correspond donc à la somme de tous les entiers naturels compris entre *2* et *n + 1* comme le montre la variable *count rajoutée à des fins de profilage*. En utilisant la formule permettant de calculer la somme des nombres naturels, on obtient:

$$c = \frac{(n+1)*(n+2)}{2} - 1 = \frac{n^2}{2} - \frac{3n}{2}$$

Par exemple, pour *n = 1000*, il faut déjà un nombre colossal d'opérations:

$$c = \frac{1000*1000}{2} + \frac{3*1000}{2} = 500000 + 1500 \approx 500000$$

Comme vous le savez, le terme quadratique l'emporte pour de grandes valeurs de *n*, ce qui explique que l'on dit que cet algorithme est **de complexité quadratique en *n***, ce que l'on note de la manière suivante :

Complexité = O(n^2)

TRIER LE JEU DE CARTES: TRI PAR INSERTION

Lorsque l'on tient un jeu de cartes en éventail, on utilise souvent de manière intuitive une autre méthode de tri : on insère chaque nouvelle carte obtenue dans l'éventail à une position bien précise correspondant à sa valeur, de sorte qu'elle soit triée par rapport aux cartes déjà présentes. Le programme suivant procède exactement de la même manière lorsqu'il pioche une carte du tas et l'insère dans la liste cible (la main) :

Il prend les cartes une à une, de gauche à droite, à partir de la liste de départ et parcourt toutes les cartes déjà insérées dans la liste de destination. Dès qu'il rencontre une carte de la liste de destination qui possède une valeur supérieure à la carte à placer, il place la nouvelle carte juste avant cette dernière carte examinée.

```
from gamegrid import *
import random

def cardValue(card):
    return card.getImage().getHeight()

def updateGrid():
   removeAllActors()
   for i in range(len(startList)):
       addActor(startList[i], Location(i, 0))
   for i in range(len(targetList)):
       addActor(targetList[i], Location(i, 1))

n = 9

makeGameGrid(n, 2, 130, Color.blue, False)
setBgColor(Color.white)
show()

startList = []
targetList = []

for i in range(0 , 9):
    card = Actor("sprites/" + "hearts" + str(i) + ".png")
    startList.append(card)

random.shuffle(startList)
updateGrid()
setTitle("Insertion Sort. Press <SPACE> to sort...")
count = 0

while not isDisposed() and len(startList) > 0:
    getBg().clear()
    c = getKeyCodeWait()
    if c == 32:
        pick = startList[0] # take first
        startList.remove(pick)
        i = 0
        while i < len(targetList) and cardValue(pick) > cardValue(targetList[i]):
            i += 1
            count += 1
        targetList.insert(i, pick)
        count += 1
        setTitle("Count: " + str(count) + " <SPACE> for next step...")
        updateGrid()
setTitle("Count: " + str(count) + " All done")
```

MEMENTO

Cette méthode de tri est appelée **tri par insertion (*insertion sort*)**. Le nombre d'opérations nécessaires dépend de l'ordre initial des cartes dans le tas. La situation qui demande le plus d'étapes survient lorsque le tas est malheureusement trié mais dans l'ordre inverse. On peut montrer, soit par une réflexion théorique soit en effectuant des simulations informatiques, que le nombre d'opérations élémentaires alors nécessaires pour effectuer un tri par insertion vaut en moyenne (pour *n* très grand), $n^2 / 4$. De ce fait, la complexité du tri par insertion est également en $O(n^2)$, comme pour le tri des enfants « *children sort* ».

TRI À BULLES (BUBBLE SORT)

Une façon assez connue de trier des objets d'une liste consiste à parcourir de manière répétée cette liste de gauche à droite et d'échanger deux objets adjacents qui sont dans le mauvais ordre. Avec cette méthode, c'est tout d'abord le plus grand élément qui va remonter toute la liste de gauche à droite pour aboutir en toute dernière position. Au prochain passage, on recommence le

même procédé tout à gauche de la liste en ne remontant cependant que jusqu'à l'avant-dernière place puisque le plus grand élément est déjà bien placé. Ce tri présente l'avantage de ne pas nécessiter de liste supplémentaire.

```
from gamegrid import *
import random

def bubbleSize(bubble):
    return bubble.getImage().getHeight()

def updateGrid():
   removeAllActors()
   for i in range(len(li)):
       addActor(li[i], Location(i, 0))

def exchange(i, j):
    temp = li[i]
    li[i] = li[j]
    li[j] = temp

n = 7
li = []

makeGameGrid(n, 1, 150, Color.red, False)
setBgColor(Color.white)
show()
for i in range(0 , n):
    bubble = Actor("sprites/bubble" + str(i) + ".png")
    li.append(bubble)
random.shuffle(li)
updateGrid()
setTitle("Bubble Sort. Press <SPACE> for next step...")
k = n - 1
i = 0
count = 0
while not isDisposed() and k > 0:
    getBg().fillCell(Location(i, 0), makeColor("beige"))
    getBg().fillCell(Location(i + 1, 0), makeColor("beige"))
    refresh()
    c = getKeyCodeWait()
    if c == 32:
        count += 1
        bubble1 = li[i]
        bubble2 = li[i + 1]
        refresh()
        if bubbleSize(bubble1) > bubbleSize(bubble2):
            exchange(i, i + 1)
            setTitle("Last Action: Exchange. Count: " + str(count))
        else:
            setTitle("Last Action: No Exchange. Count: " + str(count))
        getBg().clear()
        updateGrid()
        if i == k - 1:
            k = k - 1
            i = 0
        else:
            i += 1
getBg().clear()
refresh()
setTitle("All done. Count: " + str(count))
```

MEMENTO

Le plus grand élément se déplace progressivement de la gauche de la liste vers sa droite, exactement comme le ferait une bulle dans un verre d'eau, de bas vers le haut. C'est pour cette raison que cette méthode s'appelle **tri à bulles** (*bubble sort* en anglais). Comme vous pouvez le voir par un raisonnement ou en examinant le compteur d'opérations intégré dans le programme, la complexité de cette méthode est indépendante de l'arrangement initial des éléments mais demeure en $O(n^2)$. Pour rendre la démonstration un peu plus attractive, les deux cellules dont les bulles viennent d'être comparées sont colorées avec la méthode *fillCell()*. La couleur d'arrière-fond peut être nettoyée avec *getBg().clear().* Il est nécessaire d'invoquer la fonction *refresh()* pour faire en sorte que l'image soit réaffichée correctement à l'écran.

TRIS À L'AIDE DES FONCTIONS INTÉGRÉES: TIMSORT

Du fait que le tri est un des algorithmes les plus importants, tous les langages de programmation de haut niveau mettent à disposition des fonctions intégrées au langage permettant d'effectuer des tris. En *Python*, il s'agit de la fonction *sorted(list, cmp)* qui fait même partie de la bibliothèque de fonctions intégrées, de sorte qu'il n'est même pas nécessaire de l'importer avec *import*. Elle permet de s'économiser la tâche fastidieuse décrire un algorithme de tri. Mais il faut tout de même comprendre comment cette fonction s'utilise. Elle prend bien évidemment la liste à trier en paramètre. Le deuxième paramètre de la fonction permet de préciser le critère à utiliser pour ordonner les objets. Ce critère est défini au sein d'une fonction qui est appelée *compare() dans l'exemple ci-dessous*. Cette fonction doit accepter deux objets en guise de paramètres et retourner 1, 0, ou -1, suivant que le premier objet doit être considéré respectivement comme étant supérieur, égal ou inférieur au second objet. Cette fonction de comparaison au nom quelconque est ensuite passée en deuxième paramètre de la fonction *sorted*. Il est également possible d'utiliser le paramètre nommé *cmp*.

```
from gamegrid import *
import random

def bodyHeight(dwarf):
    return dwarf.getImage().getHeight()

def compare(dwarf1, dwarf2):
    if bodyHeight(dwarf1) < bodyHeight(dwarf2):
        return -1
    elif bodyHeight(dwarf1) > bodyHeight(dwarf2):
        return 1
    else:
        return 0

def updateGrid():
   removeAllActors()
   for i in range(len(li)):
       addActor(li[i], Location(i, 0))

n = 7
li = []

makeGameGrid(n, 1, 170, Color.red, False)
setBgColor(Color.white)
show()
for i in range(0 , n):
    dwarf = Actor("sprites/dwarf" + str(i) + ".png")
    li.append(dwarf)
random.shuffle(li)
updateGrid()
setTitle("Timsort. Press any key to get result...")
getKeyCodeWait()
li = sorted(li, cmp = compare)
```

```
updateGrid()
setTitle("All done.")
```

MEMENTO

Pour effectuer un tri à l'aide des fonctions prédéfinies dans une bibliothèque, il faut spécifier la manière dont les éléments doivent être comparés à l'aide d'une fonction de comparaison. Cela permet à la fonction *sorted* de déterminer si le premier élément d'une paire d'objets est *supérieur*, *égal* ou *inférieur* au second objet à comparer.L'algorithme utilisé en Python a été inventé par Tim Peters en 2002 et s'appelle donc *Timsort*. Sa complexité est en moyenne en O(n log(n)). De ce fait, lorsque *n* vaut par exemple 10^6, il suffit d'environ 10^7 opérations pour trier la liste au lieu des 10^{12} opérations que demanderait un algorithme de tri quadratique en $O(n^2)$.

EXERCICES

1. Trier les 7 nains à l'aide d'un tri à bulles.

2. Ajouter l'image de sprite *snowwhite.png* de Blanche Neige qui possède la même taille que le plus grand nain présent dans le tri à bulles de l'exercice 1. Montrer que l'ordre de Blanche Neige et du plus grand nain est toujours identique à l'ordre qu'ils avaient avant le début du tri. Un tel algorithme de tri est dit **stable**.

3. Il est possible de générer une liste de nombres mélangés en créant une liste de nombres triés *row = range(n)* et en les mélangeant ensuite avec *random.shuffle(row)*. Mesurer le temps d'exécution de l'algorithme intégré à Python (*Timsort*) pour différentes valeurs de *n* et montrer que sa complexité est bien meilleure que $O(n^2)$.
   Indication : Pour mesurer une différence de temps, il faut importer le module *time* et calculer la différence de temps qui s'est écoulée entre deux appels successifs à la fonction *time.clock()*.

ATÉRIEL SUPPLÉMENTAIRE

REDÉFINIR LES OPÉRATIONS DE COMPARAISON

La comparaison de deux objets constitue une opération très importante. Les nombres admettent les cinq opérateurs de comparaison <, <=, ==, >, = >. En *Python, il est possible d'appliquer ces opérateurs de comparaison à d'autres objets comme les nains par exemple, ce qui fait gagner au code en élégance et en clarté.* Voici comment procéder:

Dans la classe qui définit les objets à trier, il faut définir les méthodes *__lt__(), __le__(), __eq__(), __ge__(), __gt__()* qui retournent la valeur booléenne de l'opérateur de comparaison correspondant *less, less-and-equal, equal, greater-and-equal, greater*.

De plus, il est également possible de redéfinir la méthode *__str()__*, qui sera automatiquement appelée par Python lorsque l'on veut convertir l'objet en chaine de caractères avec la fonction *str()*. Cela n'a cependant rien à voir avec le tri. Dans la classe *Dwarf* (dérivée de *Actor*), on peut également stocker le nom du nain dans une variable d'instance que l'on peut représenter à l'écran en tant que *TextActor* lors de l'exécution de la fonction *updateGrid()*.

```
from gamegrid import *
import random

class Dwarf(Actor):
    def __init__(self, name, size):
        Actor.__init__(self, "sprites/dwarf" + str(size) + ".png")
        self.name = name
        self.size = size
    def __eq__(self, a):  # ==
        return self.size == a.size
    def __ne__(self, a): # !=
        return self.size != a.size
    def __gt__(self, a): # >
        return self.size > a.size
    def __lt__(self, a): # <
        return self.size < a.size
    def __ge__(self, a): # >=
        return self.size >= a.size
    def __le__(self, a): # <=
        return self.size <= a.size
    def __str__(self):  # str() function
        return self.name

def compare(dwarf1, dwarf2):
    if dwarf1 < dwarf2:
        return -1
    elif dwarf1 > dwarf2:
        return 1
    else:
        return 0

def updateGrid():
   removeAllActors()
   for i in range(len(row)):
       addActor(row[i], Location(i, 0))
       addActor(TextActor(str(row[i])), Location(i, 0))

n = 7
row = []
names = ["Monday", "Tuesday", "Wednesday", "Thursday",
         "Friday", "Saturday", "Sunday"]

makeGameGrid(n, 1, 170, Color.red, False)
setBgColor(Color.white)
show()
for i in range(0 , n):
    dwarf = Dwarf(names[i], i)
    row.append(dwarf)
random.shuffle(row)
updateGrid()
setTitle("Press any key to get result...")
getKeyCodeWait()
row = sorted(row, cmp = compare)
updateGrid()
setTitle("All done.")
```

MEMENTO

L'usage d'opérateurs de comparaison personnalisés pour un type de données arbitraire n'est pas obligatoire mais constitue une solution élégante pour permettre d'en trier les instances. On dit que l'on a redéfini ou **surchargé** l'opérateur (*overloaded* en anglais).

10.2 PROBLÈMES INSOLUBLES

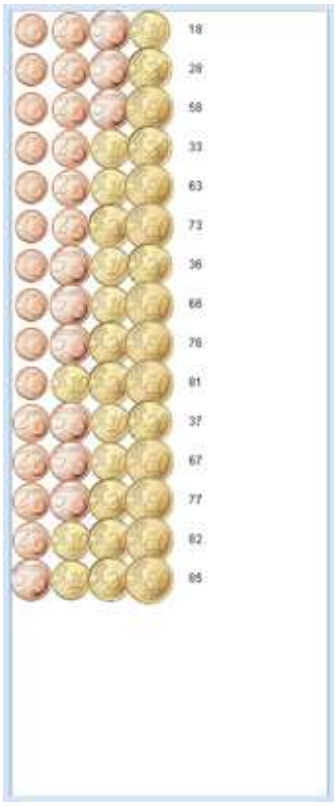
INTRODUCTION

ILe nombre de problèmes qu'il est possible de résoudre par ordinateur est de plus en plus important. Dans ce chapitre, nous serons cependant confrontés à des problèmes qui se laissent formuler très aisément mais qui ne pourront sans doute jamais être résolus de manière algorithmique. Il y a de fortes chances pour que cela ne change pas malgré la croissance rapide de la puissance de traitement des ordinateurs et toutes les recherches scientifiques qui se consacrent à ces problèmes.

CONCEPTS DE PROGRAMMATION: *Problèmes insolubles, problème de la somme des sous-ensembles, méthode d'énumération, explosion combinatoire, ordre polynomial, problème indécidable*

PROBLÈMES INSOLUBLES

Il reste encore aujourd'hui de nombreux problèmes non résolus alors même qu'ils sont très faciles à énoncer et qu'ils présentent un intérêt conséquent en pratique. L'un de ces problèmes, nommé le problème de la somme des sous-ensembles, peut être formulé de la manière suivante [**plus...**]:



Vous disposez dans votre porte-monnaie d'un certain nombre de pièces de monnaie et vous devez payer une certaine somme à un automate sans qu'il ne rende la monnaie. Cela est-il possible avec les pièces dont vous disposez et, si oui, quelles sont les pièces qu'il faut utiliser?

Notre premier programme nous permettra d'apprendre à gérer les pièces de monnaie. Le programme commence par stocker dans la liste *coins* le nom des pièces d'Euro de valeur 1, 2, 5, 10, 20 et 50 centimes. La fonction *value()* retourne la valeur de la pièce. Le porte-monnaie est modélisé par une liste (ou un tuple) *moneybag* contenant le nom des pièces présentes dans le porte-monnaie. La fonction *getSum(moneybag) retourne la valeur totale de l'ensemble des pièces se trouvant dans le portemonnaie.*

Dans un premier temps, prenons un porte-monnaie contenant exactement un exemplaire de chaque pièce. Le programme construit alors toutes les différentes combinaisons de pièces comportant 1, 2, 3, 4, 5 ou 6 pièces et les affiche dans une fenêtre *JGameGrid*. Pour ce faire, la fonction *showMoneybag(moneybag, y) crée* une instance de la classe *Actor* pour chaque pièce de monnaie du porte-monnaie et l'affiche dans la fenêtre sur la rangée (ligne) *y*.

```
from gamegrid import *
import itertools

coins = ["one", "two", "five", "ten", "twenty", "fifty"]

def value(coin):
```

```
    if coin == "one":
        return 1
    if coin == "two":
        return 2
    if coin == "five":
        return 5
    if coin == "ten":
        return 10
    if coin == "twenty":
        return 20
    if coin == "fifty":
        return 50
    return 0

def getSum(moneybag):
    sum = 0
    for coin in moneybag:
        sum += value(coin)
    return sum

def showMoneybag(moneybag, y):
    x = 0
    for coin in moneybag:
        loc = Location(x, y)
        removeActor(getOneActorAt(loc))
        coinActor = Actor("sprites/" + coin + "cent.png")
        addActor(coinActor, loc)
        x += 1
    addActor(TextActor(str(getSum(moneybag))), Location(x, y))

makeGameGrid(8, 20, 40, False)
setBgColor(Color.white)
show()

n = 6
k = 1
while k <= n:
    combinations = list(itertools.combinations(coins, k))
    print type(combinations)
    setTitle("(n, k) = (" + str(n) + ", " + str(k) + ") nb = "
    + str(len(combinations)))
    y = 0
    for moneybag in combinations:
        showMoneybag(moneybag, y)
        y += 1
    getKeyCodeWait()
    removeAllActors()
    k += 1
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

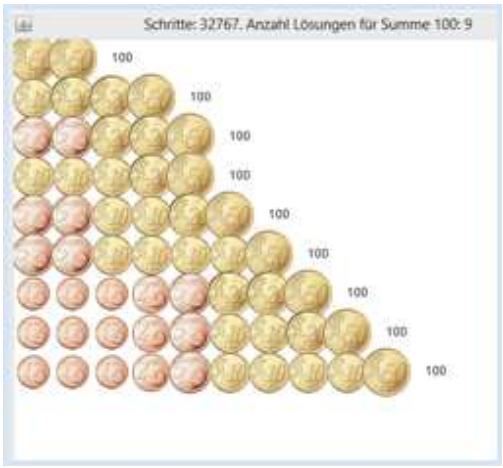
MEMENTO

La fonction *combinations()* du module *itertools* permet d'obtenir facilement toutes les combinaisons de *k* éléments que l'on peut fabriquer à partir des éléments d'une liste de longueur *n*. Il est cependant nécessaire de convertir la valeur de retour en une liste pour en extraire une à une chacune des combinaisons sous forme de tuple.

Les combinaisons ainsi obtenues sont ordonnées selon un ordre naturel semblable à celui que l'on obtiendrait si l'on avait fait le travail à la main. On peut calculer exactement le nombre de combinaisons de longueur *k* issues d'un ensemble de *n* éléments grâce au fameux coefficient binomial :

$$c = \binom{n}{k} = \frac{n!}{k!*(n-k)!}$$

où n! est la factorielle de *n,* à savoir le produit de tous les nombres de 1 à *n*. Pour *n=6*, on pourrait avoir 6, 15, 20, 15, 6, 1 et, de ce fait, un total de 63 combinaisons



On peut résoudre le problème de la somme des sous-ensembles du porte-monnaie de la manière suivante : il faut déterminer toutes les combinaisons possibles de pièces de monnaies présentes dans le porte-monnaie et tester si la somme de ce sous-ensemble correspond à la somme désirée.

Cette méthode d'énumération n'est probablement pas la plus efficace que l'on puisse imaginer mais elle a le mérite d'être correcte et de fournir toutes les solutions possibles. Pour un porte-monnaie qui contient 3 pièces de 1 ct, 1 pièce de 2 ct, 2 pièces de 5 ct, 4 pièces de 10 ct, 2 pièces de 20 ct et 3 pièces de 50 ct (15 pièces en tout), il serait déjà difficile de trouver la solution à la main par énumération. On n'écrit que les combinaisons dont la somme totale se monte à un Euro .

```
from gamegrid import *
import itertools

coins = ["one", "one", "one", "two", "five", "five",
         "ten", "ten", "ten", "ten", "twenty", "twenty",
         "fifty", "fifty", "fifty"]

def value(coin):
    if coin == "one":
        return 1
    if coin == "two":
        return 2
    if coin == "five":
        return 5
    if coin == "ten":
        return 10
    if coin == "twenty":
        return 20
    if coin == "fifty":
        return 50
    return 0

def getSum(moneybag):
    sum = 0
    for coin in moneybag:
        sum += value(coin)
    return sum

def showMoneybag(moneybag, y):
    x = 0
    for coin in moneybag:
        loc = Location(x, y)
        removeActor(getOneActorAt(loc))
        coinActor = Actor("sprites/" + coin + "cent.png")
        addActor(coinActor, loc)
        x += 1
```

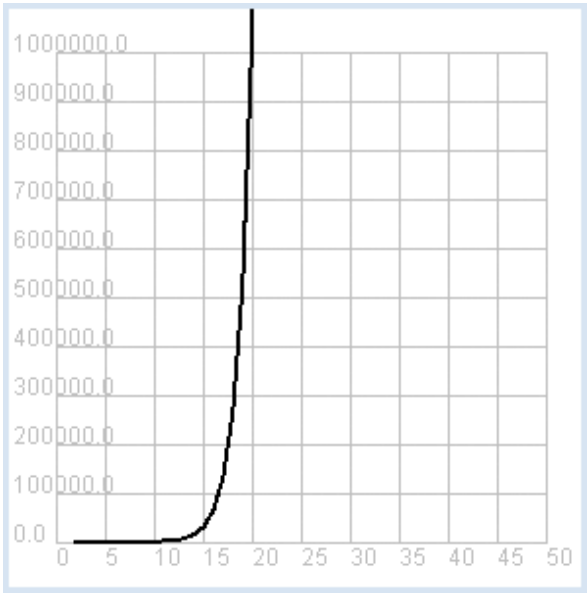
```
    addActor(TextActor(str(getSum(moneybag))), Location(x, y))

makeGameGrid(15, 20, 40, False)
setBgColor(Color.white)
show()

target = 100

k = 1
result = []
count = 0
while k <= len(coins):
    combinations = tuple(itertools.combinations(coins, k))
    nb = len(combinations)
    for moneybag in combinations:
        count += 1
        sum = getSum(moneybag)
        if sum == target:
            if not moneybag in result:
                result.append(moneybag)
    k += 1

y = 0
for moneybag in result:
    showMoneybag(moneybag, y)
    y += 1
setTitle("Step: " + str(count) + ". number of solutions for the sum  "
         + str(target) + ": " + str(len(result)))
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Pour un nombre restreint de 15 pièces de monnaie, une méthode par énumération nécessite déjà la bagatelle de 32'767 étapes pour résoudre le problème de la somme des sous-ensembles.

On peut être tout fou d'être en mesure de développer un programme qui s'acquittera de cette tâche très rapidement mais on déchantera rapidement lorsque l'on sera confronté à un nombre légèrement supérieur de pièces de monnaies, par exemple 50 ou 100. Si l'on compte le nombre de pas nécessaires pour un porte-monnaie comptant *n* pièces et que l'on affiche ce résultat dans un graphique lorsque *n* augmente, on constate qu'il y a une véritable explosion combinatoire pour *n=20* qui dépasse tout ce qui est imaginable avec les ordinateurs actuels. **plus...**].



```
from gpanel import *
from math import factorial

z = 100

def nbCombi(n, k):
    return factorial(n) / factorial(k) / factorial(n - k)

makeGPanel(-5, 55, -1e5, 1.1e6)
drawGrid(0, 50, 0, 1e6, "gray")
setColor("black")
lineWidth(2)
for n in range(2, z + 1):
    sum = 0
    for k in range(1, n):
        sum += nbCombi(n, k)
    print "n =", n, ", nb =", sum
    if n == 2:
        move(n, sum)
    else:
        draw(n, sum)
print "Runtime with 10^9 operations per second:", sum / 3.142e16, "years"
print "or:", int(sum / 2e20), "times the age of the universe"
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Si l'on utilise la méthode de l'énumération, le problème de la somme des sous-ensembles est **déjà insoluble** pour un nombre relativement faible d'éléments, alors même que l'algorithme de résolution est connu. Il reste encore à savoir s'il n'existerait pas des algorithmes **qualitativement très supérieurs** dont la complexité temporelle serait une puissance de *n* (complexité polynomiale) comme le sont les algorithmes de tri vus dans le chapitre précédent. Malheureusement, personne n'a jusqu'à présent trouvé un tel algorithme pour le problème de la somme des sous-ensembles et on part en général du principe qu'il n'y en pas. Par contre, il n'existe pas non plus de preuve qu'un tel algorithme n'existe pas.

On sait du moins de l'informatique théorique qu'il existe de nombreux problèmes de la même classe de difficulté et que si l'on trouve une méthode efficace de résolution pour l'un de ces problèmes, alors tous les problèmes de cette difficulté sont d'emblée résolubles à partir de cette méthode [**plus...**].

PROBLÈMES INDÉCIDABLES

Les limites de l'esprit humain et de la technologie informatique se révèlent également dans un contexte différent de celui de la théorie de la complexité. Le mathématicien et théoricien des nombres Lothar Collatz s'est penché sur certaines suites de nombres et a formulé en 1939 la question suivante:

Prenons une suite de nombres dont le terme initial est un nombre naturel quelconque dont les termes consécutifs sont construits à partir des règles de récurrence suivantes :

- Si *n* est pair, diviser *n* par 2 (qui est à nouveau un nombre naturel)
- Si *n* est impair, prendre le nombre *3n+1* (qui est forcément un nombre pair)

Question : Cette suite converge-t-elle toujours vers 1 quel que soit le terme initial *n*?

Collatz ainsi que de nombreux autres théoriciens des nombres et chercheurs en informatique ont tenté de répondre à cette question puisque même les plus puissants ordinateurs de la planète obtiennent sans arrêt des suites qui atteignent le nombre 1 (les suites ne convergent

pas car elles répètent de manière infinie la séquence 4, 2, 1).

Il apparaît donc **vraisemblable** que le théorème suivant soit vérifié:

*Pour tout terme initial n, l*a suite *3n+1 atteint le nombre 1 en un nombre fini d'étapes.*

On peut faire soi-même l'expérience et parcourir la suite (3n+1) à l'aide d'un programme informatique pour un nombre initial *n* quelconque.

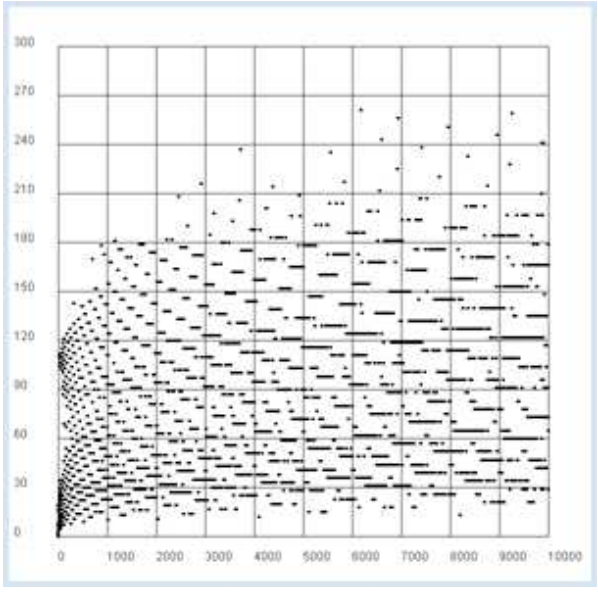
```
from gpanel import *

def collatz(n):
    while n != 1:
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
        print n,
    print "Result 1"
while True:
    n = inputInt("Enter a start number:")
    collatz(n)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

En Python, il est même possible de calculer les termes de la suite 3n+1 pour un terme initial très grand. Selon le théorème précédent, la suite en question va toujours finir, après un nombre suffisamment grand mais fini d'itérations, par tomber sur le nombre 1. Évidemment, ceci ne constitue aucunement une preuve de la question posée par Collatz.

Il est intéressant et même très esthétique de représenter la longueur de la suite 3n+1 en fonction du terme initial de la suite. On remarque que cette longueur fluctue considérablement. Pour ce faire, il faut supprimer l'affichage dans la console des termes de la suite au sein de la fonction *collatz()* et se contenter de retourner le nombre d'étapes jusqu'à ce que l'on tombe sur 1.




```
from gpanel import *

def collatz(n):
    nb = 0
    while n != 1:
        nb += 1
        if n % 2 == 0:
            n = n // 2
```

```
        else:
            n = 3 * n + 1
    return nb

z = 10000 # max n
yval = [0] * (z + 1)
for n in range(1, z + 1):
    yval[n] = collatz(n)
ymax = (max(yval) // 100  + 1) * 100

makeGPanel(-0.1 * z, 1.1 * z, -0.1 * ymax, 1.1 * ymax)
title("Collatz Assumption")
drawGrid(0, z, 0, ymax, "gray")

for x in range(1, z + 1):
    move(x, yval[x])
    fillCircle(z / 200)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

L'hypothèse de Collatz est un problème vraiment très difficile. En supposant que l'hypothèse soit vraie, il n'est pas possible de la prouver en effectuant un très grand nombre de tests par ordinateurs pour des nombres *n* toujours plus grands. Il est même possible que l'hypothèse soit vraie mais qu'il ne soit pas possible de prouver sa véracité. En 1931, le mathématicien Kurt Gödel a montré avec son théorème d'incomplétude qu'il peut exister des affirmations qui sont vraies à l'intérieur d'une théorie mais dont la véracité ne peut pas être prouvée.

Le problème de Collatz peut également être formulé comme un **problème de décision**:

Un algorithme qui calcule les termes de la suite 3n+1 et qui s'arrête à 1 s'arrête-t-il vraiment pour tous les termes initiaux possibles?

On peut essayer de résoudre cette question par ordinateur. Malheureusement, cette tentative est probablement complètement vaine elle aussi car le grand mathématicien Alan Turing a déjà prouvé avec son problème de l'arrêt (**Halting Problem** en anglais) qu'il n'existera jamais un algorithme général permettant de décider si un programme va s'arrêter quelles que soient les données qu'on lui fournit en entrée.

Il se peut donc que l'hypothèse de Collatz soit correcte mais qu'elle constitue un **problème indécidable**.

10.3 RETOUR SUR TRACE (BACKTRACKING)

INTRODUCTION

Dans le domaine du développement de jeu vidéo, les choses commencent à devenir intéressantes lorsque l'ordinateur devient lui-même un joueur présentant une certaine intelligence. Un tel programme doit non seulement se conformer aux règles du jeu mais également poursuivre une stratégie de victoire. Afin d'implémenter une stratégie de jeu, il faut voir le jeu comme une séquence de situations pouvant être clairement identifiés à l'aide d'une variable appropriée *s*. Ces situations sont appelées **états du jeu**, raison pour laquelle la variable *s* est appelée **variable d'état**. La stratégie gagnante consiste à passer de l'état initial à un état gagnant dans lequel le jeu est terminé.

On peut se représenter les états du jeu comme des **nœuds** dans le **graphe de jeu**. À chaque tour de jeu, il y a une transition qui se fait entre le nœud actuel et l'un de ses successeurs. Les règles du jeu spécifient quels sont les **nœuds successeurs** possibles pour l'état actuel, également appelés **nœuds voisins** du nœud actuel. On peut représenter cette transition par une ligne orientée (flèche), également appelée **arête orientée**. [**plus...**].

Dans cette section, nous allons aborder des techniques importantes qui sont valables de manière générale et qui vous aideront à développer des jeux vidéo capables de gagner même contre des joueurs humains très intelligents. Il vous restera cependant encore beaucoup de liberté et d'occasions pour développer vos propres idées ainsi que des stratégies de jeu plus efficaces, plus simples et mieux adaptées qui permettront d'améliorer le jeu encore davantage ou de consommer moins de puissance de calcul.

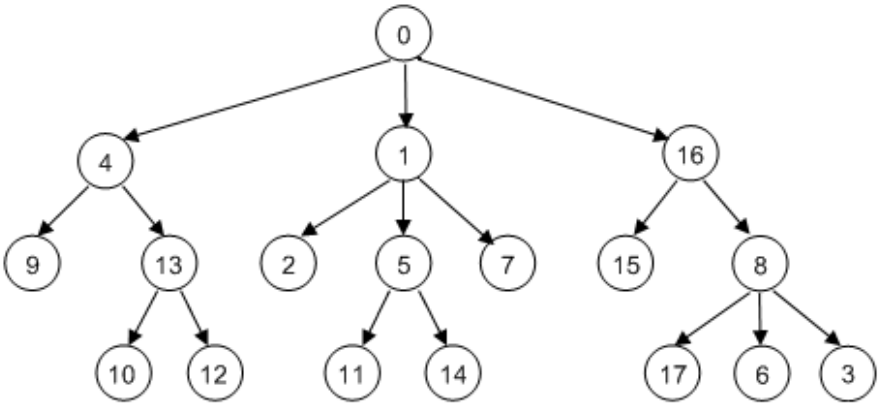
De plus, il est un fait que de nombreux systèmes politiques, économiques ou sociaux peuvent être compris et modélisés comme des jeux, ce qui vous permettra d'appliquer les notions abordées ci-après dans un panel très large de domaines particulièrement pertinents.

CONCEPTS DE PROGRAMMATION: *Game state, game graph, depth-first search, backtracking*

RECHERCHE D'UNE SOLUTION POUR UN JEU EN SOLITAIRE

Comme nous l'avons vu précédemment, on peut représenter les états de jeu comme des nœuds dans un graphe qui est parcouru pas à pas. Il faut commencer par déterminer de manière unique les états du jeu selon certains critères tels que l'arrangement des pièces sur le plateau de jeu. Les règles du jeu spécifient quels sont les successeurs (ou voisins) possibles d'un état du jeu particulier. Ces nœuds sont alors reliés dans le graphe par une arête. Puisqu'il s'agit d'états successeurs, les arêtes sont orientées en partant du nœud représentant l'état actuel vers ses successeurs. Un nœud est parfois appelé « père » et ses voisins des « fils » et il est possible qu'il y ait une arête orientée partant de fils pour remonter au père.

Prenons d'abord un graphe simple issu d'un jeu se jouant en solo ou par une seule personne contre l'ordinateur. Le jeu en solitaire devrait être conçu de telle sorte qu'il ne soit pas possible de revenir à un état antérieur. Cela garantit que le graphe est exempt de cycle dans lequel on pourrait tourner en rond à l'infini en traversant le graphe. Un tel cas particulier de graphe est appelé **arbre** [**plus...**]. On peut identifier les nœuds dans n'importe quel ordre par des nombres compris entre 0 et 17. Le graphe présente la structure suivante:



Il faut pouvoir stocker l'arbre en entier dans une structure de données appropriée. Une bonne idée est d'utiliser une liste dont chaque élément d'indice *i* est une sous-liste comprenant les nœuds fils du nœud *i*. Ainsi, les fils du nœud 0 se trouveront dans la sous-liste placée à la position 0, les fils du nœud 1 figureront dans la sous-liste placée à la position *1*, etc … Si un nœud ne possède pas de fils, la liste des nœuds fils correspondante sera vide [**plus...**].

L'arbre illustré ci-dessus peut donc être représenté par la liste suivante:

```
neighbours = [[4, 1, 16], [2, 5, 7], [], [], [9, 13],[11, 14], [], [], [17, 6,
3], [], [], [], [], [10, 12], [], [], [15, 8], []]
```

Le fait d'identifier un nœud par un nombre est une astuce qui permet de déterminer facilement ses nœuds fils au sein de la liste grâce à l'indice. L'algorithme permettant de trouver le chemin pour passer d'un nœud à un autre nœud placé plus en profondeur dans l'arbre est défini récursivement dans la fonction *search(node)*. Voici sa formulation en **pseudo code** :

```
search(node):
   if node == targetnode:
        print "Target achieved"
        return
   Déterminer la liste neighbors des voisins de node
   parcourir cette liste et faire pour chacune:
        search(neighbors)
```

De plus, les nœuds visités sont ajoutés à la fin de la liste *visited*. Si l'état visé n'est pas atteint avant, le dernier nœud ajouté à la liste *visited* en est supprimé après que tous ses fils ont été visités. Cela permet de restaurer l'état en vigueur avant la visite infructueuse de ce nœud [**plus...**]. Le nœud de départ et d'arrivée peuvent être saisis au début de l'exécution du programme.

```
neighbours = [[4, 1, 16], [2, 5, 7], [], [], [9, 13], [11, 14], [], [],
              [17, 6, 3], [], [], [], [], [10, 12], [], [], [15, 8], []]

def search(node):
    visited.append(node) # put (push) to stack

    # Check for solution
    if node == targetNode:
        print "Target ", targetNode, "achieved. Path:", visited
        targetFound = True
        return

    for neighbour in neighbours[node]:
        search(neighbour) # recursive call
    visited.pop() # redraw (pop) from stack

startNode = -1
while startNode < 0 or startNode > 17:
   startNode = inputInt("Start node (0..17):")
targetNode = -1
```

```
while targetNode < 0 or targetNode > 17:
   targetNode = inputInt("Target node (0..17):")
visited = []
search(startNode)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

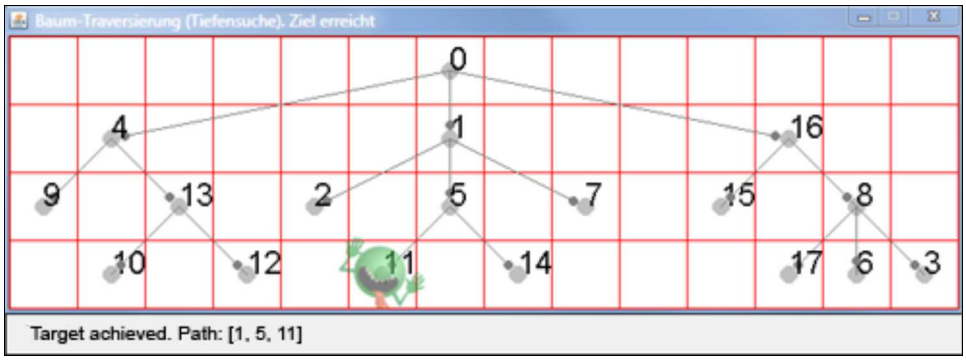
MEMENTO

Le chemin correct [0, 1, 5, 14] pour le nœud de départ 0 et le nœud cible 14 est imprimé dans la sortie standard. Si l'on rajoute le nœud 0 comme voisin du nœud 13, on introduit un cycle dans le graphe et il en résulte une situation catastrophique et le programme se termine avec une exception indiquant que la profondeur maximale de récursion a été atteinte.

LA TRAVERSÉE D'UN ALIEN

Il serait agréable de pouvoir visualiser l'exécution de l'algorithme en représentant graphiquement l'arbre de décision du jeu et en le traversant progressivement au fur et à mesure du déroulement en pressant sur une touche. Pour ce faire, utilisons une fenêtre *GameGrid* dans laquelle les nœuds de l'arbre sont représentés par un cercle placé au milieu d'une cellule de la grille (coordonnées grille / grid location).

Un alien semi-transparent met en évidence l'état courant du jeu.



On dessine l'arbre en utilisant les méthodes graphiques de la classe *GGBackground*. Pour montrer la direction des arêtes, il est possible d'ajouter un petit cercle aux arêtes en lieu et place d'une pointe de flèche en utilisant la méthode *getMarkerPoint(). Il faut alors s'assurer de rafraîchir l'écran avec refresh()*. La barre d'état montre des informations importantes.

```
from gamegrid import *

neighbours = [[4, 1, 16], [2, 5, 7], [], [], [9, 13], [11, 14], [], [],
              [17, 6, 3], [], [], [], [], [10, 12], [], [], [15, 8], []]

locations = [Location(6, 0), Location(6, 1), Location(4, 2), Location(13, 3),
             Location(1, 1), Location(6, 2), Location(12, 3), Location(8, 2),
             Location(12, 2), Location(0, 2), Location(1, 3), Location(5, 3),
             Location(3, 3), Location(2, 2), Location(7, 3), Location(10, 2),
             Location(11, 1), Location(11, 3)]

def drawGraph():
    getBg().clear()
    for i in range(len(locations)):
        getBg().setPaintColor(Color.lightGray)
        getBg().fillCircle(toPoint(locations[i]), 6)
        getBg().setPaintColor(Color.black)
        getBg().drawText(str(i), toPoint(locations[i]))
        for k in neighbours[i]:
            drawConnection(i, k)
    refresh()
```

```
def drawConnection(i, k):
    getBg().setPaintColor(Color.gray)
    startPoint = toPoint(locations[i])
    endPoint = toPoint(locations[k])
    getBg().drawLine(startPoint, endPoint)
    getBg().fillCircle(getMarkerPoint(endPoint, startPoint, 10), 3)

def search(node):
    global targetFound
    if targetFound:
        return
    visited.append(node) # put (push) to stack
    alien.setLocation(locations[node])
    refresh()
    if node == targetNode:
        setStatusText("Target " + str(targetNode) + "achieved. Path: "
                      + str(visited))
        targetFound = True
        return
    else:
        setStatusText("Current node " + str(node) + " .  Visited: "
        + str(visited))
    getKeyCodeWait(True) # exit if GameGrid is disposed

    for neighbour in neighbours[node]:
        search(neighbour)  # Recursive call
    visited.pop()

makeGameGrid(14, 4, 50, Color.red, False)
setTitle("Tree-traversal (depth-first search). Press a key...")
addStatusBar(30)
show()
setBgColor(Color.white)
drawGraph()

startNode = -1
while startNode < 0 or startNode > 17:
   startNode = inputInt("Start node  (0..17):")
targetNode = -1
while targetNode < 0 or targetNode > 17:
   targetNode = inputInt("Target node  (0..17):")

visited = []
targetFound = False
alien = Actor("sprites/alieng_trans.png")
addActor(alien, locations[startNode])

search(startNode)
setTitle("Tree-traversal (depth-first search). Target achieved")
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Comme vous pouvez le constater, l'alien se déplace vers les nœuds fils « en profondeur d'abord » et remonte au dernier nœud parent lorsque tous les fils ont été visités. C'est pour cette raison que cet algorithme s'appelle « **recherche en profondeur avec retour sur trace** ». (***depth-first search with backtracking***).

L'ALIEN SUR LE CHEMIN DU RETOUR

Afin de rendre visible le chemin emprunté par l'alien lors de son retour sur trace, il est nécessaire de sauvegarder la séquence des nœuds visités lors du parcours en profondeur. Le passage à

chaque niveau de profondeur supplémentaire engendre une nouvelle liste que l'on sauvegarde dans *stepsList*. Après avoir effectué un retour sur trace, il faut supprimer la dernière entrée de cette liste avec *stepsList.pop()*.

```
from gamegrid import *

neighbours = [[4, 1, 16], [2, 5, 7], [], [], [9, 13], [11, 14], [], [],
              [17, 6, 3], [], [], [], [], [10, 12], [], [], [15, 8], []]

locations = [Location(6, 0), Location(6, 1), Location(4, 2), Location(13, 3),
             Location(1, 1), Location(6, 2), Location(12, 3), Location(8, 2),
             Location(12, 2), Location(0, 2), Location(1, 3), Location(5, 3),
             Location(3, 3), Location(2, 2), Location(7, 3), Location(10, 2),
             Location(11, 1), Location(11, 3)]

def drawGraph():
    getBg().clear()
    for i in range(len(locations)):
        getBg().setPaintColor(Color.lightGray)
        getBg().fillCircle(toPoint(locations[i]), 6)
        getBg().setPaintColor(Color.black)
        getBg().drawText(str(i), toPoint(locations[i]))
        for k in neighbours[i]:
            drawConnection(i, k)
    refresh()

def drawConnection(i, k):
    getBg().setPaintColor(Color.gray)
    startPoint = toPoint(locations[i])
    endPoint = toPoint(locations[k])
    getBg().drawLine(startPoint, endPoint)
    getBg().fillCircle(getMarkerPoint(endPoint, startPoint, 10), 3)



def search(node):
    global targetFound
    if targetFound:
        return
    visited.append(node) # put (push) to stack
    alien.setLocation(locations[node])
    refresh()
    if node == targetNode:
        setStatusText("Target " + str(targetNode) + "achieved. Path: "
                      + str(visited))
        targetFound = True
        return
    else:
        setStatusText("Current nodes " + str(node) + " .  Visited: "
                      + str(visited))
    getKeyCodeWait(True) # exit if GameGrid is disposed

    for neighbour in neighbours[node]:
        steps = [node]
        stepsList.append(steps)
        steps.append(neighbour)
        search(neighbour)  # Recursive call
        steps.reverse()
        if not targetFound:
            for loc in steps[1:]:
                setStatusText("Go back")
                alien.setLocation(locations[loc])
                refresh()
                getKeyCodeWait()
        stepsList.pop()
    visited.pop()

makeGameGrid(14, 4, 50, Color.red, False)
setTitle("Tree-traversal (depth-first search). Press a key...")
```

```
addStatusBar(30)
show()
setBgColor(Color.white)
drawGraph()

startNode = -1
while startNode < 0 or startNode > 17:
   startNode = inputInt("Start node  (0..17):")
targetNode = -1
while targetNode < 0 or targetNode > 17:
   targetNode = inputInt("Target node (0..17):")

visited = []
stepsList = []
targetFound = False
alien = Actor("sprites/alieng_trans.png")
addActor(alien, locations[startNode])

search(startNode)
setTitle("Tree-traversal (depth-first search). Target  achieved")
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

À présent, l'alien revient véritablement sur ses traces dans l'arbre, ce qui rend particulièrement évident le fait que cet algorithme s'appelle **retour sur trace** (*backtracking*). Le retour sur trace récursif joue un rôle très important dans de nombreux algorithmes, à tel point qu'il est parfois considéré comme « le couteau suisse du programmeur ».

STATÉGIE DANS UN LABYRINTHE

Il est parfois très difficile, voire même angoissant, de parvenir à trouver le chemin menant à la sortie d'un labyrinthe. Heureusement, grâce à vos connaissances sur le retour sur trace, vous êtes maintenant en mesure d'écrire un programme capable de trouver la sortie à tous les coups pour autant que celle-ci existe. Il est relativement évident qu'il est possible de modéliser un labyrinthe ne possédant pas de cycle par un arbre. Il apparaît donc que de trouver la sortie d'un tel labyrinthe correspond en fait à traverser un arbre.

Dans le cas présent, nous allons nous contenter d'un petit labyrinthe aléatoire de 11x11 cellules. L'alien se déplace d'une seule étape à chaque pression d'une touche du clavier mais recherche directement la sortie de manière autonome lors d'une pression sur la touche Enter.

Le labyrinthe est généré à l'aide de la classe *Maze* dont le constructeur prend en argument des entiers impairs représentant le nombre de lignes et de colonnes désirées. À chaque fois, la classe génère un labyrinthe aléatoire différent dont l'entrée se trouve en haut à gauche et la sortie en bas à droite. La méthode *isWall(loc)* permet de tester si la position *loc* correspond à un mur.

Il n'est souvent pas souhaitable de déterminer le graphe de jeu complet avant le début du jeu. Très souvent, cela est même impossible puisqu'il y aurait tant de situations à prévoir qu'il faudrait un temps CPU et un stockage dans la RAM complètement démesuré. De ce fait, il est souvent préférable de ne déterminer les nœuds fils du nœud courant que lors du parcours de l'arbre.

En l'occurrence, on détermine les nœuds voisins du nœud actuel en sélectionnant les cellules adjacentes (au maximum 4) qui ne sont pas des murs et qui ne se situent pas en dehors de la grille.

```
from gamegrid import *

def createMaze():
    global maze
    maze = GGMaze(11, 11)
    for x in range(11):
        for y in range(11):
            loc = Location(x, y)
            if maze.isWall(loc):
                getBg().fillCell(loc, Color(0, 50, 0))
            else:
                getBg().fillCell(loc, Color(255, 228, 196))
    refresh()

def getNeighbours(node):
    neighbours = []
    for loc in node.getNeighbourLocations(0.5):
        if isInGrid(loc) and not maze.isWall(loc):
           neighbours.append(loc)
    return neighbours

def search(node):
    global targetFound, manual
    if targetFound:
        return
    visited.append(node)  # push
    alien.setLocation(node)
    refresh()
    delay(500)
    if manual:
        if getKeyCodeWait(True) == 10:  #Enter
            setTitle("Finding target...")
            manual = False

    # Check for termination
    if node == exitLocation:
        targetFound = True

    for neighbour in getNeighbours(node):
        if neighbour not in visited:
            backSteps = [node]
            backStepsList.append(backSteps)
            backSteps.append(neighbour)

            search(neighbour) # recursive call

            backSteps.reverse()
            if not targetFound:
                for loc in backSteps[1:]:
                    setTitle("Must go back...")
                    alien.setLocation(loc)
                    refresh()
                    delay(500)
                if manual:
                    setTitle("Went back. Press key...")
                else:
                    setTitle("Went back. Find target...")
            backStepsList.pop()
    visited.pop() # pop

manual = True
targetFound = False
visited = []
backStepsList = []
makeGameGrid(11, 11, 40, False)
setTitle("Press a key. (<Enter> for automatic")
show()
createMaze()
```

```
startLocation = maze.getStartLocation()
exitLocation = maze.getExitLocation()
alien = Actor("sprites/alieng.png")
addActor(alien, startLocation)
search(startLocation)
setTitle("Target found")
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

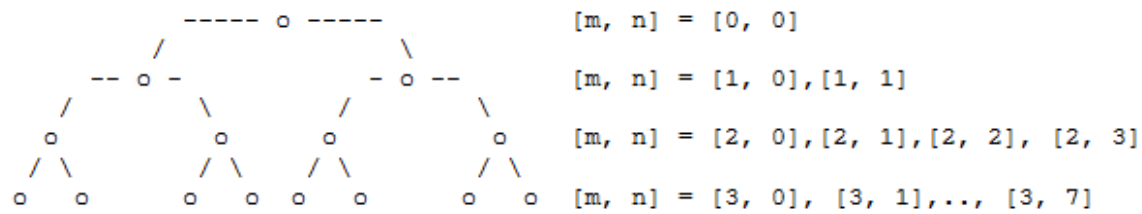
MEMENTO

Il est intéressant de comparer la stratégie de résolution d'un humain avec celle de l'ordinateur. Un joueur humain est capable de saisir la situation d'ensemble en un coup d'œil et d'en déduire une stratégie permettant de trouver la sortie de manière très directe. Il utilise pour cela une aptitude de vision globale caractéristique de l'humain dont l'ordinateur est totalement dépourvu. L'ordinateur, quant à lui, ne possède qu'une vision très locale de la situation en lien avec sa position actuelle mais il a l'avantage de « se souvenir » exactement de tous les chemins déjà empruntés et peut donc chercher de manière très systématique de nouveaux chemins jusqu'à tomber fatalement sur celui qui le mènera à la sortie.

L'humain est donc favorisé s'il possède une vue d'avion du labyrinthe mais le vent tourne en faveur de l'ordinateur si l'humain est lui-même restreint à une vision locale en se trouvant dans le labyrinthe, pour autant que les murs soient trop haut pour qu'il puisse voir « au-delà » du couloir actuel.

EXERCISES

1. Dans un arbre binaire, chaque nœud possède exactement deux nœuds fils, à savoir un nœud gauche et un nœud droit. Représentons chacun des nœuds par une liste de deux entiers [m,n] où *m* symbolise la profondeur du nœud au sein de l'arbre et *n* sa position en largeur.



   Développer un programme qui écrit le chemin permettant de passer d'un nœud initial à un nœud final.

2. Il est remarquable qu'il soit toujours possible de trouver la sortie d'un labyrinthe en utilisant la règle de la main droite, même lorsque le labyrinthe comporte des cycles. Cette règle consiste à suivre scrupuleusement le mur avec la main droite en observant les règles suivantes :

   - Si la cellule à droite de la cellule actuelle est libre (pas de mur), bouger vers cette cellule
   - Si ce n'est pas possible (la cellule à droite est un mur), continuer tout droit
   - Si aucune des deux premières possibilités n'est possible, tourner à gauche

   Implémenter cette règle pour le labyrinthe *GameGrid* en utilisant une coccinelle pivotable comme acteur avec *lady = Actor(True, "sprites/ladybug.gif").* Indications : Placer la coccinelle dans la cellule selon la règle de la main droite en utilisant la fonction *move()*. Si la cellule en question est un mur, annuler le mouvement.

   Comparer cette solution avec celle obtenue par l'algorithme de retour sur trace.

MATÉRIAL SUPPLÉMENTAIRE

LE PROBLÈME DES *N* DAMES

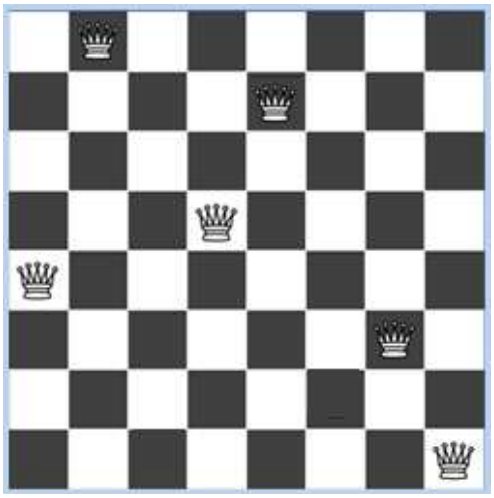
Le problème des *N* dames est un problème d'échecs qui a déjà été discuté depuis le milieu du 19e siècle. Il s'agit de placer *N* dames sur un échiquier de taille *N* fois *N* de telle sorte qu'elles ne se menacent pas mutuellement selon les règles habituelles des échecs qui stipulent que la dame se peut se déplacer horizontalement, verticalement et en diagonale. Ce problème appelle deux types de solutions dont les niveaux de difficulté sont radicalement différents. La solution la plus facile à obtenir consiste à trouver un placement particulier des dames qui satisfait les contraintes du problème. L'autre type de solution, beaucoup plus difficile à atteindre, consiste à déterminer le nombre de placements possibles. Le mathématicien Glaisher a prouvé en 1874 déjà que le problème des *N* dames comportait 92 solutions pour un échiquier habituel, à savoir pour *N=8*.

Le problème des *N* dames est considéré comme un candidat classique pour la résolution par retour sur trace. Il s'agit de placer les dames une à une sur l'échiquier de telle sorte que chaque nouvelle dame placée ne menace aucune dame placée précédemment. Si l'on procède de manière naïve, il y a tôt ou tard un moment où l'on ne peut plus placer de dame. La stratégie du retour sur trace consiste alors à supprimer la dernière dame placée pour tenter une alternative. Si cette nouvelle tentative ne mène toujours pas à une solution, il faut encore retirer une dame de plus et ainsi de suite, jusqu'à ce qu'une solution soit trouvée. Un humain s'emmêlerait vite les pinceaux en perdant le contrôle des solutions déjà testées mais l'ordinateur ne souffre pas de ce problème car il procède de manière purement mécanique et méthodique.

De même que pour toute situation à traiter par retour sur trace, on peut considérer les états de jeu comme des nœuds dans le graphe de jeu. Il est à cet effet crucial de choisir une structure de données appropriée. Procéder par la force brute en testant toutes les façons possibles de placer les *N* dames sur l'échiquier et en rejetant celles où les dames s'attaquent mutuellement est totalement impensable. En effet, lorsque *N* vaut seulement 8, il y a déjà environ 422 millions placements possibles à tester.

Il serait bien plus malin de ne considérer dès le début que des positions dans lesquelles chaque dame se trouve seule sur sa colonne et sa rangée. À ce dessein, pour *N=8*, on peut représenter l'état du jeu en se contentant d'une liste de 8 nombres dont l'élément *i* représente la position de la dame *i* dans la rangée *i*. Les lignes dépourvues de dame sont représentées par le nombre -1.

En utilisant des indices de ligne et de colonne compris entre 0 et n-1, le placement représenté ci-contre serait représenté par la liste *node = [1, 4, -1, 3, 0, 6, -1, 7].*



On peut déterminer les noeuds voisins du nœud actuel (*node*) au sein de l'algorithme de retour sur trace à l'aide de la fonction *getNeighbours(node)*. On passe de ce fait d'une structure de données unidimensionnelle à la liste *Locations* qui utilise les coordonnées *x* et *y* des cases de l'échiquier. Les cases déjà occupées sont référencées dans la liste *squares* et celles qui ne peuvent pas être utilisées en vertu des règles des échecs dans la liste *forbidden*. Il est à cet effet utile de recourir à la méthode *getDiagonalLocations()*. Finalement, on crée la liste *allowed* pour stocker les cases qui peuvent encore recevoir une dame. Il faut à ce stade remplacer le -1 dans la liste des nœuds voisins par l'indice de la colonne sur laquelle la nouvelle dame a été placée.

L'algorithme de retour sur trace maintenant bien connu est implémenté dans la fonction *search()*. Dès qu'une solution a été trouvée, la recherche par retour sur trace est arrêtée (condition d'arrêt de la

récursion).

```
from gamegrid import *

n = 8 # number of queens

def getNeighbours(node):
    squares = [] # list of occupied squares
    for i in range(n):
        if node[i] != -1:
           squares.append(Location(node[i], i))

    forbidden = [] # list of forbidden squares
    for location in squares:
        a = location.x
        b = location.y
        for x in range(n):
            forbidden.append(Location(x, b))  # same row
        for y in range(n):
            forbidden.append(Location(a, y))  # same column
        for loc in getDiagonalLocations(location, True):   #diagonal up
            forbidden.append(loc)
        for loc in getDiagonalLocations(location, False):  #diagonal down
            forbidden.append(loc)

    allowed = [] # list of all allowed squares = all - forbidden
    for i in range(n):
        for k in range(n):
            loc = Location(i, k)
            if not loc in forbidden:
                allowed.append(loc)

    neighbourNodes = []
    for loc in allowed:
        neighbourNode = node[:]
        i = loc.y # row
        k = loc.x # col
        neighbourNode[i] = k
        neighbourNodes.append(neighbourNode)
    return neighbourNodes

def search(node):
    global found
    if found or isDisposed():
        return
    visited.append(node)  # node marked as visited

    # Check for solution
    if not -1 in node:
        found = True
        drawNode(node)

    for s in getNeighbours(node):
        search(s)
    visited.pop()

def drawBoard():
    for i in range(n):
        for k in range(n):
            if (i + k) % 2 == 0:
                getBg().fillCell(Location(i, k), Color.white)

def drawNode(node):
    removeAllActors()
    for i in range(n):
        addActorNoRefresh(Actor("sprites/chesswhite_1.png"), Location(node[i], i))
    refresh()

makeGameGrid(n, n, 600 // n, False)
setBgColor(Color.darkGray)
```

```
drawBoard()
show()
setTitle("Searching. Please wait..." )

visited = []
found = False
startNode = [-1] * n  # all squares empty
search(startNode)
setTitle("Search complete. ")
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Suivant les performances de votre ordinateur, la solution devrait émerger en quelques secondes ou en quelques minutes. Si l'exécution est trop lente, il suffit de réduire la taille du problème en posant *N=6*.

EXERCICES

1. Résoudre la même tâche sans utiliser la bibliothèque *GameGrid*. Remplacer les objets *Location* par une liste de listes [i, k] représentant la position de la case dans l'échiquier. La solution peut être imprimée dans la console comme une liste de nœuds..

2. Généraliser le programme réalisé précédemment pour N = 6 ou votre propre programme de l'exercice 1 pour qu'il trouve toutes les solutions possibles. Remarquez que certains nœuds reviennent à plusieurs reprises (symétries et rotations de l'échiquier). Éviter les doublons en maintenant une liste de solutions déjà trouvées.

10.4 PLUS COURT CHEMIN, PROBLÈME DES 3 RÉCIPIENTS

INTRODUCTION

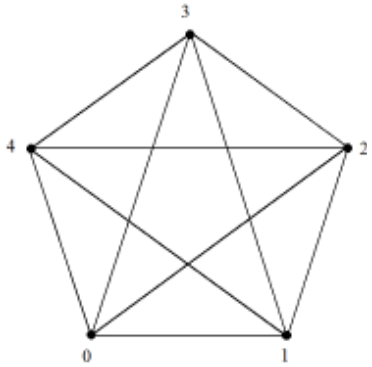
De nombreuses solutions algorithmiques ont été développées à partir de l'expérience quotidienne, y compris le retour sur trace. Comme vous l'avez compris, l'ordinateur choisit un coup parmi tous les coups permis et poursuit ainsi de manière parfaitement mécanique et cohérente. Si l'ordinateur se retrouve bloqué dans une impasse, il annule les coups précédents. Cette stratégie s'apparente au tâtonnement dans le contexte de la vie quotidienne et il est bien connu qu'elle n'est souvent pas optimale du tout. Il serait bien préférable de choisir le prochain coup le plus intelligent possible pour se rapprocher du but recherché [**plus...**].

CONCEPTS DE PROGRAMMATION: Tâtonnement, graphe comportant des cycles, retour sur trace

GRAPHES COMPORTANT DES CYCLES

Lors du parcours d'un graphe, il peut arriver que l'on se retrouve en un nœud qui a déjà été visité quelques étapes plus tôt. Prenons l'exemple du réseau de métro souterrain d'une grande métropole dont les gares comportent de nombreuses liaisons différentes. Ce réseau permet d'atteindre une gare A à partir d'une gare B de différentes manières et l'on pourrait rapidement se retrouver à tourner en rond sans jamais atteindre la gare souhaitée. Prenons l'exemple du graphe suivant comportant 5 nœuds tous reliés deux à deux par une arête.

Les nœuds sont identifiés par les nombres 0 à 4. Comme vous l'aurez compris, il est fort probable que le simple algorithme de retour sur trace ne parvienne pas à trouver le chemin reliant A à B s'il reste bloqué dans un cycle. Il suffit cependant d'un remède facile à implémenter : il suffit de vérifier que le prochain voisin que l'algorithme s'apprête à visiter ne figure pas encore dans la liste *visited* avant d'effectuer l'appel récursif à *search()*. S'il est présent, il faut simplement ignorer ce nœud et continuer l'algorithme avec le nœud suivant. Avec cette correction, tous les 16 chemins possibles entre A et B sont imprimés dans la console.




```
def getNeighbours(node):
    return range(0, node) + range(node + 1, 5)

def search(node):
    global nbSolution
    visited.append(node)  # node marked as visited

    # Check for solution
    if node == targetNode:
        nbSolution += 1
        print nbSolution, ". Route:", visited
        # don't stop to get all solutions

    for neighbour in getNeighbours(node):
        if neighbour not in visited: # Check if already visited
            search(neighbour) # recursive call
    visited.pop()

startNode = 0
targetNode = 4
```

```
nbSolution = 0
visited = []
search(startNode)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

En vérifiant au préalable si un nœud a déjà été visité, il est possible d'utiliser le retour sur trace dans un graphe comportant des cycles. Si l'on oublie d'effectuer cette vérification, le programme va se planter et lever une erreur d'exécution due à un débordement de la pile d'appels récursifs (appels récursifs infinis).

PLUS COURT CHEMIN, SYSTÈME DE NAVIGATION

Chercher son chemin pour passer d'un point A à un point de destination B est un problème récurrent dans la vie quotidienne. Comme il y a de nombreux chemins qui mènent de A à B (et non seulement à Rome), il est également important de déterminer un certain critère (longueur du chemin, durée du voyage, qualité de la route, paysages, coûts etc…) dans le but de trouver le chemin optimal [**plus...**].

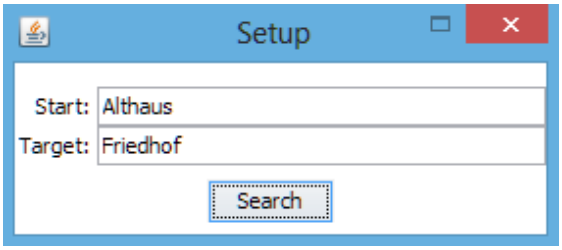
Dans le programme développé ci-après, on se concentre sur les bases. Il traite le cas d'un réseau de métro ne comportant que 6 arrêts dans une ville fictive. Les nœuds du graphe sont identifiés par le nom des stations qui commencent par les lettres A, B, C, D, E et F. On aurait donc également pu identifier les stations par les lettres A, B, C, D, E, F ou par un numéro de station. Pour stocker les relations de voisinage entre stations, on utilise un dictionnaire dont chaque élément est constitué d'une station en guise de clé et de la liste de toutes les stations qui lui sont adjacentes dans le réseau en guise de valeur. Au lieu d'utiliser une fonction *getNeighbours()*, ont fait directement usage d'un dictionnaire *neighbours*.

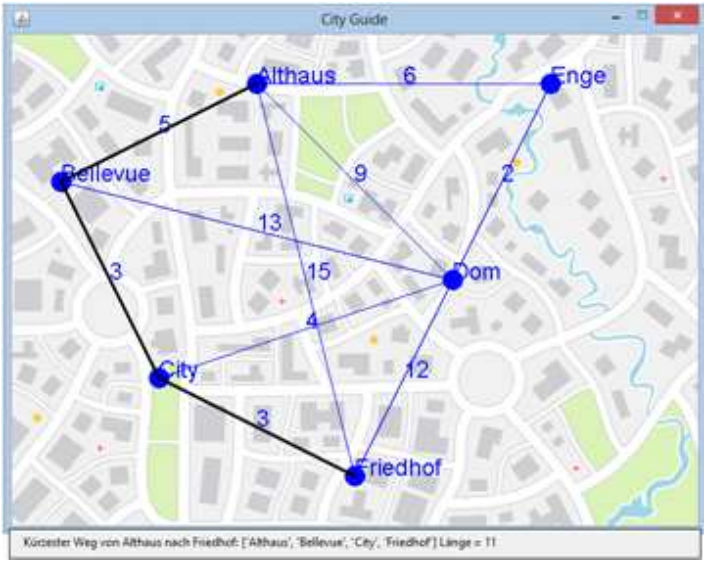
De manière similaire, les distances séparant les stations sont stockées dans un autre dictionnaire *distances* analogue au dictionnaire *neighbours*.

Il faut également recourir à un dictionnaire *locations pour stocker les coordonnées x* et *y* des emplacements des stations.

L'essentiel du programme est une copie exacte de l'algorithme de retour sur trace utilisé précédemment. De plus, il faut également quelques fonctions auxiliaires pour permettre la représentation graphique.

On peut utiliser une boîte de dialogue pour récupérer la saisie utilisateur et écrire la sortie dans la barre d'état. De plus, le programme dessine le chemin optimal dans le graph reliant les stations.






```
from gamegrid import *

neighbours = {
 'Althaus':['Bellevue', 'Dom', 'Enge'],
 'Bellevue':['Althaus', 'City', 'Dom'],
 'City':['Bellevue', 'Dom', 'Friedhof'],
 'Dom':['Althaus', 'Bellevue', 'City', 'Enge', 'Friedhof'],
 'Enge':['Althaus', 'Dom'],
 'Friedhof':['Althaus', 'City', 'Dom']}

distances = {
   ('Althaus', 'Bellevue'):5, ('Althaus', 'Dom'):9,
   ('Althaus', 'Enge'):6, ('Althaus', 'Friedhof'):15,
   ('Bellevue', 'City'):3, ('Bellevue', 'Dom'):13,
   ('City', 'Dom'):4, ('City', 'Friedhof'):3,
   ('Dom', 'Enge'):2, ('Dom', 'Friedhof'):12}

locations = {
 'Althaus':Location(2, 0),
 'Bellevue':Location(0, 1),
 'City':Location(1, 3),
 'Dom':Location(4, 2),
 'Enge':Location(5, 0),
 'Friedhof':Location(3, 4)}

def getNeighbourDistance(station1, station2):
    if station1 < station2:
        return distances[(station1, station2)]
    return distances[(station2, station1)]

def totalDistance(li):
    sum = 0
    for i in range(len(li) - 1):
        sum += getNeighbourDistance(li[i], li[i + 1])
    return sum

def drawGraph():
    getBg().clear()
    getBg().setPaintColor(Color.blue)
    for station in locations:
        location = locations[station]
        getBg().fillCircle(toPoint(location), 10)
        startPoint = toPoint(location)
        getBg().drawText(station, startPoint)
        for s in neighbours[station]:
            drawConnection(station, s)
            if s < station:
                distance = distances[(s, station)]
            else:
```

```
                distance = distances[(station, s)]
            endPoint = toPoint(locations[s])
            getBg().drawText(str(distance),
                   getDividingPoint(startPoint, endPoint, 0.5))
    refresh()

def drawConnection(startStation, endStation):
    startPoint = toPoint(locations[startStation])
    endPoint = toPoint(locations[endStation])
    getBg().drawLine(startPoint, endPoint)

def search(station):
    global trackToTarget, trackLength
    visited.append(station)  # station marked as visited

    # Check for solution
    if station == targetStation:
        currentDistance = totalDistance(visited)
        if currentDistance < trackLength:
            trackLength = currentDistance
            trackToTarget = visited[:]

    for s in neighbours[station]:
        if s not in visited:  # if all are visited, recursion returns
            search(s) # recursive call
    visited.pop() # station may be visited by another path

def init():
    global visited, trackToTarget, trackLength
    visited = []
    trackToTarget = []
    trackLength = 1000
    drawGraph()

makeGameGrid(7, 5, 100, None, "sprites/city.png", False)
setTitle("City Guide")
addStatusBar(30)
show()
init()
startStation = ""
while not startStation in locations:
    startStation = inputString("Start station")
targetStation = ""
while not targetStation in locations:
    targetStation = inputString("Target station")
search(startStation)
setStatusText("Shortest way from " + startStation + " to " + targetStation
    + ": " + str(trackToTarget) + " Length = " + str(trackLength))
for i in range(len(trackToTarget) - 1):
    s1 = trackToTarget[i]
    s2 = trackToTarget[i + 1]
    getBg().setPaintColor(Color.black)
    getBg().setLineWidth(3)
    drawConnection(s1, s2)
refresh()
```

MEMENTO

La recherche du plus court chemin au sein d'un graph constitue l'une des tâches élémentaires de la science de l'information. La solution que nous avons étudiée pour déterminer le plus court chemin atteint son but par retour sur trace mais présente le désavantage d'être extrêmement gourmande en ressources CPU. Il existe de bien meilleurs algorithmes pour trouver le plus court chemin qui évitent d'explorer systématiquement tous les chemins possibles. Le plus fameux d'entre eux est appelée "algorithme de Dijkstra".

LE PROBLÈME DES TROIS JERRICANS

Les casse-têtes consistant à diviser une quantité donnée en une fraction d'elle-même par mesure (en pesant, versant, etc …) fourmillent depuis des lustres dans les livres pour enfants et dans les magazines. Le célèbre problème des trois récipients est attribué au mathématicien français du 17e siècle Bachet de Méziriac et s'énonce comme suit:

Deux amis décident de partager de manière équitable huit litres de vin contenus dans un récipient de huit litres par vidage et transferts. Ils disposent également d'un récipient de cinq litres et d'un autre de trois litres tous dépourvus de graduation. Comment doivent-ils procéder et combien de transferts faut-il au minimum?

D'après la formulation du problème, il ne s'agit pas seulement de trouver la solution, ce qui est tout à fait faisable en réfléchissant la moindre, mais de déterminer toutes les solutions possibles et de trouver la plus courte. Sans ordinateur, cette tâche serait très fastidieuse. Rechercher toutes les solutions possibles comme dans le cas présent constitue une **recherche exhaustive**.

Pour commencer, on procède par retour sur trace, ce qui nécessite l'élaboration d'une structure de données appropriées pour stocker les états du jeu. Pour ce faire, on utilisera une liste de trois nombres décrivant le niveau de remplissage de chacun des récipients. L'état [1, 4, 3] signifie donc que le récipient de 8 litres contient 1 litre de vin, le récipient de 5 litres en contient 4 et le récipient de 3 litres en contient 3.

Une fois n'est pas coutume, on modélise les états du jeu par des nœuds dans un graphe et l'on considère le transfert des récipients vers un autre commune une transition d'un nœud du graphe vers un nœud qui lui est adjacent. Dans cette situation, comme dans tant d'autres, il est insensé de vouloir construire dès le début tout l'arbre de décision. Il est en effet préférable de ne déterminer les nœuds voisins du nœud actuel *node* par la fonction *getNeighbours(node) que lorsque cela est nécessaire. Il faut partir de la considération suivante*:

Peu importe la quantité de vin présente dans un récipient, il y a essentiellement 6 options différentes de transfert d'un récipient à l'autre. Pour chaque récipient, on peut soit vider totalement son contenu dans un autre récipient ou ne transférer que la quantité de vin correspondant au volume encore disponible dans le récipient de destination. On rassemble donc les nœuds correspondant à ces 6 possibilités de transfert dans la liste *neighbours au sein de la fonction getNeighbours()*. Pour déterminer les nœuds voisins de l'état courant, la fonction *transfer(state, source, target)* retourne l'état résultant d'un transfert du récipient *source* vers *target.* On considère pour ce faire autant la contenance maximale des récipients que le volume de vin qu'ils contiennent déjà. Encore une fois, la fonction récursive *search()* utilise l'algorithme de retour sur trace.

```
def transfer(state, source, target):
    # Assumption: source, target 0..2, source != target
    s = state[:] # clone
    if s[source] == 0 or s[target] == capacity[target]:
        return s  # source empty or target full
    free = capacity[target] - s[target]
    if s[source] <= free: # source has enough space in target
        s[target] += s[source]
        s[source] = 0
    else:  # target is filled-up
        s[target] = capacity[target]
        s[source] -= free
    return s

def getNeighbours(node):
```

```
# returns list of neighbours
    neighbours = []
    t = transfer(node, 0, 1) # from 0 to 1
    if t not in neighbours:
       neighbours.append(t)
    t = transfer(node, 0, 2) # from 0 to 2
    if t not in neighbours:
        neighbours.append(t)
    t = transfer(node, 1, 0) # from 1 to 0
    if  t not in neighbours:
        neighbours.append(t)
    t = transfer(node, 1, 2) # from 1 to 2
    if t not in neighbours:
        neighbours.append(t)
    t = transfer(node, 2, 0) # from 2 to 0
    if t not in neighbours:
        neighbours.append(t)
    t = transfer(node, 2, 1) # from 2 to 1
    if t not in neighbours:
        neighbours.append(t)
    return neighbours

def search(node):
    global nbSolution
    visited.append(node)

    # Check for solution
    if node == targetNode:
        nbSolution += 1
        print nbSolution, ". Route:", visited, ". Length:", len(visited)

    for s in getNeighbours(node):
        if s not in visited:
            search(s)
    visited.pop()

capacity = [8, 5, 3]
startNode = [8, 0, 0]
targetNode = [4, 4, 0]
nbSolution = 0
visited = []
search(startNode)
print "Done. Find the best  solution!"
```

MEMENTO

Les solutions sont écrites dans la console et ne figurent pas dans ce texte pour vous permettre d'appréhender le problème sans *a priori* et de tenter de trouver les solutions par vous-mêmes avec un papier et un crayon. Après les avoir découvertes, vous constaterez qu'il y a 16 solutions dont la plus longue nécessite 16 transferts.

EXERCISES

1. Simplifier le programme de navigation de telle sorte que les nœuds soient identifiés par les nombres 0, 1, 2, 3, 4, 5 et que *neighbours* soit une liste contenant des sous-listes de nœuds.

2. Décrire le processus permettant de puiser d'un lac exactement 4 litres d'eau à l'aide d'un récipient de 3 litres et d'un récipient de 5 litres. Trouver la solution demandant le moins de transferts possibles. N'oubliez pas qu'il est possible de vider un récipient dans le lac

3. Inventer un casse-tête de transfert dans le style de l'exercice précédent en vous assurant qu'il possède une solution. Demandez à vos amis de le résoudre.

MATÉRIAL SUPPLÉMENTAIRE

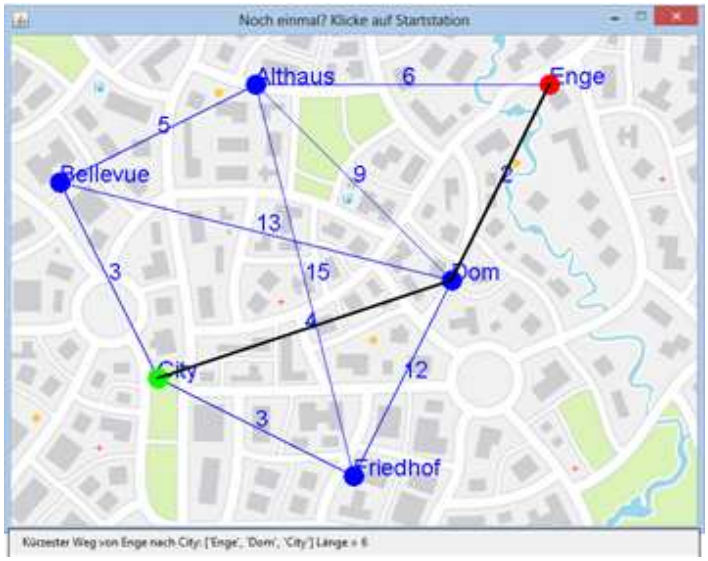
SYSTÈME DE NAVIGATION URBAIN À L'AIDE DE LA SOURIS

L'expérience utilisateur (UX = User eXperience) joue un rôle primordial dans n'importe quelle application professionnelle. Lors de la phase de conception, le développeur ou le concepteur ne doit pas tant se laisser guider par des considérations d'ordre techniques du domaine de la programmation mais bien plutôt en se mettant dans la peau d'un utilisateur lambda qui veut pouvoir utiliser l'application de la manière la plus intuitive possible et en ne recourant qu'à son bon sens. De nos jours, cela implique des interfaces intégrant une surface contrôlable à l'aide de la souris ou de gestes tactiles. Le développement de l'interface utilisateur nécessite souvent une partie non négligeable de l'effort global investi dans un projet de développement informatique.

Les systèmes de navigation adoptent de plus en plus les interfaces tactiles. Celles-ci adoptent cependant une logique relativement similaire au pilotage par la souris. De ce fait, nous allons modifier notre système de navigation urbain pour ajouter le support de la souris, de telle sorte que l'utilisateur puisse sélectionner le point de départ et celui d'arrivée à l'aide d'un clic de souris. Les résultats seront écrits aussi bien dans la barre de titre de la fenêtre que dans sa barre d'état.

Le clic de la souris engendre un événement qui est géré dans la fonction de rappel *pressEvent()* que l'on enregistre avec la fonction *makeGameGrid()* grâce au paramètre nommé *mousePressed*. Il faut garder à l'esprit que le programme peut se trouver dans deux états différents suivant que l'utilisateur a déjà effectué un premier clic pour sélectionner le point de départ ou non. S'il a déjà effectué ce premier clic, le prochain clic servira à déterminer le point de destination souhaité. Il suffira de recourir un fanion booléen *isStart pour distinguer ces deux états possibles. Ce dernier sera mis à True si le prochain clic est censé permettre la sélection du point de départ du trajet.*

L'interface utilisateur devrait être conçue de telle manière qu'il soit possible de sélectionner plusieurs trajets d'affilée sans avoir à redémarrer le programme. De ce fait, il est nécessaire que le programme soit en mesure de retourner à un état initial bien défini. Ce processus qui est mené à bien par la fonction *init()* est appelé **réinitialisation**. Du fait que certains éléments sont initialisés automatiquement au démarrage du système, il n'est en aucun cas trivial d'implémenter à l'aide de fonctions définies par le programme lui-même un système de réinitialisation personnalisé permettant de retourner de manière répétée à un état initial propre. Les erreurs d'initialisation sont de ce fait très courantes en programmation en plus d'être généralement fâcheuses et difficiles à identifier puisque le programme se comporte souvent correctement durant les phases de test et ne commence à montrer des signes de dysfonctionnement que plus tard, lorsqu'il est mis en production.



```
from gamegrid import *

locations = {
 'Althaus':Location(2, 0),
 'Bellevue':Location(0, 1),
 'City':Location(1, 3),
 'Dom':Location(4, 2),
 'Enge':Location(5, 0),
 'Friedhof':Location(3, 4)}

neighbours = {
 'Althaus':['Bellevue', 'Dom', 'Enge'],
 'Bellevue':['Althaus', 'City', 'Dom'],
 'City':['Bellevue', 'Dom', 'Friedhof'],
 'Dom':['Althaus', 'Bellevue', 'City', 'Enge', 'Friedhof'],
 'Enge':['Althaus', 'Dom'],
 'Friedhof':['Althaus', 'City', 'Dom']}

distances = {('Althaus', 'Bellevue'):5, ('Althaus', 'Dom'):9,
             ('Althaus', 'Enge'):6, ('Althaus', 'Friedhof'):15,
             ('Bellevue', 'City'):3, ('Bellevue', 'Dom'):13,
             ('City', 'Dom'):4, ('City', 'Friedhof'):3,
             ('Dom', 'Enge'):2, ('Dom', 'Friedhof'):12}

def getNeighbourDistance(station1, station2):
    if station1 < station2:
        return distances[(station1, station2)]
    return distances[(station2, station1)]

def totalDistance(li):
    sum = 0
    for i in range(len(li) - 1):
        sum += getNeighbourDistance(li[i], li[i + 1])
    return sum

def drawGraph():
    getBg().clear()
    getBg().setPaintColor(Color.blue)
    for station in locations:
        location = locations[station]
        getBg().fillCircle(toPoint(location), 10)
        startPoint = toPoint(location)
        getBg().drawText(station, startPoint)
        for s in neighbours[station]:
            drawConnection(station, s)
            if s < station:
                distance = distances[(s, station)]
            else:
                distance = distances[(station, s)]
            endPoint = toPoint(locations[s])
            getBg().drawText(str(distance),
                 getDividingPoint(startPoint, endPoint, 0.5))
    refresh()

def drawConnection(startStation, endStation):
    startPoint = toPoint(locations[startStation])
    endPoint = toPoint(locations[endStation])
    getBg().drawLine(startPoint, endPoint)

def search(station):
    global trackToTarget, trackLength
    visited.append(station)  # station marked as visited

    if station == targetStation:
        currentDistance = totalDistance(visited)
        if currentDistance < trackLength:
            trackLength = currentDistance
            trackToTarget = visited[:]
```

```
    for s in neighbours[station]:
        if s not in visited:  # if all are visited, recursion returns
            search(s) # recursive call
    visited.pop() # station may be visited by another path

def getStation(location):
    for station in locations:
        if locations[station] == location:
            return station
    return None # station not found

def init():
    global visited, trackToTarget, trackLength
    visited = []
    trackToTarget = []
    trackLength = 1000
    drawGraph()

def pressEvent(e):
    global isStart, startStation, targetStation
    mouseLoc = toLocationInGrid(e.getX(), e.getY())
    mouseStation = getStation(mouseLoc)
    if mouseStation == None:
        return
    if isStart:
        isStart = False
        init()
        setTitle("Click on destination  station")
        startStation = mouseStation
        getBg().setPaintColor(Color.red)
        getBg().fillCircle(toPoint(mouseLoc), 10)
    else:
        isStart = True
        setTitle("Once again? Click on  starting station.")
        targetStation = mouseStation
        getBg().setPaintColor(Color.green)
        getBg().fillCircle(toPoint(mouseLoc), 10)
        search(startStation)
        setStatusText("Shortest route from " + startStation + " to "
            + targetStation + ": " + str(trackToTarget) + " Length = "
            + str(trackLength))
        for i in range(len(trackToTarget) - 1):
            s1 = trackToTarget[i]
            s2 = trackToTarget[i + 1]
            getBg().setPaintColor(Color.black)
            getBg().setLineWidth(3)
            drawConnection(s1, s2)
            getBg().setLineWidth(1)
    refresh()

isStart = True
makeGameGrid(7, 5, 100, None, "sprites/city.png", False,
             mousePressed = pressEvent)
setTitle("City Guide. Click on starting  station.")
addStatusBar(30)
show()
init()
```

MEMENTO

La partie algorithmique impliquant le retour sur trace demeure pratiquement inchangée. L'interface utilisateur impliquant le contrôle avec la souris est par contre relativement complexe malgré les facilités mises à disposition par les fonctions de rappel. L'usage du mot-clé *global* peut facilement conduire à des erreurs d'initialisation du fait que les variables globales peuvent être créées depuis certaines fonctions et que l'on pourrait facilement oublier de les réinitialiser.

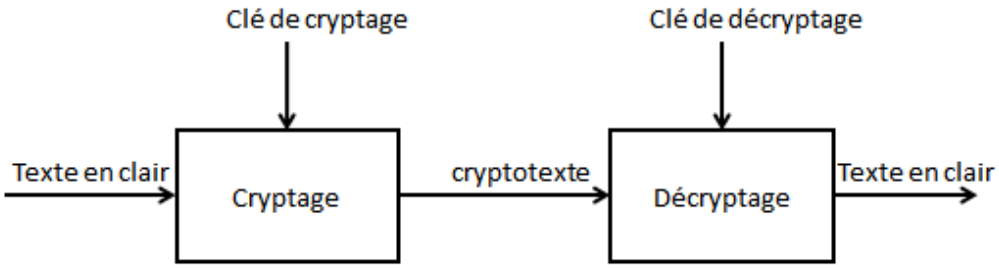
10.5 CRYPTOSYSTÈMES

INTRODUCTION

Le principe du secret des données joue un rôle de plus en plus important dans notre société moderne car elle garantit le respect de la vie privée ainsi que la confidentialité des informations gouvernementales, industrielles ou militaires. Pour cela, il est nécessaire de crypter les données de telle sorte que si elles venaient à tomber entre les mains des fausses personnes, il soit impossible ou du moins très difficile de retrouver l'information originale sans que la méthode de cryptage soit compromise au préalable.

Durant **l'encodage**, les données originales sont transformées en données codées et lors du **décodage**, les données originales sont restaurées à partir des données cryptées. Si les données originales sont constituées de lettres de l'alphabet, on parle de **texte en clair** et de **cryptotexte**.

La description de la méthode utilisée pour effectuer le décryptage est appelée **clé** qui peut consister simplement en un nombre, une chaine de caractères numériques alphabétiques. Si la même clé est utilisée pour le codage et le décodage, on parle de **système à clé symétrique**. Si, en revanche, les clés de codage et de décodage sont différentes, on parle de méthode de **cryptographie asymétrique** (à clé publique).



CONCEPTS DE PROGRAMMATION: *Encodage, décodage, cryptographie symétrique/asymétrique, Chiffre de César, Chiffre de Vigenère, cryptage RSA, clé rivée/publique*

CHIFFRE DE CÉSAR

D'après la tradition, Jules César (100 av. J.-C. - 44 av. J.-C.) utilisait déjà la méthode suivante dans ses communications militaires : chaque caractère du texte en clair était décalé dans l'alphabet d'un certain nombre de lettres en reprenant au début de l'alphabet une fois arrivé au bout.

Cette méthode utilisait deux anneaux imbriqués portant les lettres de l'alphabet. L'anneau intérieur était décalé d'un certain nombre de positions, ce qui constitue la clé de cryptage. Par exemple, si la clé vaut 3, le A sera codé par un D, le B par un E, le C par un F, le D par un G etc …

Le programme ci-dessous lit le message à crypter depuis un fichier texte pour permettre de le modifier et de le partager facilement. Il faut écrire le texte en clair à l'aide d'un éditeur de code standard dans le fichier *original.txt* qu'il faut sauvegarder dans le même dossier que le programme

Python. Il faut restreindre le message original aux lettres capitales, aux espaces et aux retours à la ligne. On aura par exemple

ON SE RETROUVE AUJOURD HUI A HUIT HEURES
BISOUS
TANIA

La fonction *encode(msg)* encode la chaine de caractères *msg* du message issu du fichier texte. Elle procède en remplaçant chaque caractère original par le caractère crypté correspondant, excepté le caractère de retour à la ligne *\n*.

```
import string
key = 4
alphabet = string.ascii_uppercase + " "

def encode(text):
    enc = ""
    for ch in text:
        if ch != "\n":
            i = alphabet.index(ch)
            ch = alphabet[(i + key) % 27]
        enc += ch
    return enc

fInp = open("original.txt")
text = fInp.read()
fInp.close()

print "Original:\n", text
krypto = encode(text)
print "Krypto:\n", krypto

fOut = open("secret.txt", "w")
for ch in krypto:
    fOut.write(ch)
fOut.close()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

Voici le texte crypté:

SRDWIDVIXVSYZIDEYNSYVHDLYMDEDLYMXDLIYVIW
FMWSYW
XERME

Le décodage est implémenté de façon analogue à part le fait que les caractères sont décalés dans l'autre sens

```
import string
key = 4
alphabet = string.ascii_uppercase + " "

def decode(text):
    dec = ""
    for ch in text:
        if ch != "\n":
            i = alphabet.index(ch)
            ch = alphabet[(i - key) % 27]
        dec += ch
    return dec

fInp = open("secret.txt")
krypto = fInp.read()
fInp.close()

print "Krypto:\n", krypto
```

```
msg = decode(krypto)
print "Message:\n", msg

fOut = open("message.txt", "w")
for ch in msg:
    fOut.write(ch)
fOut.close()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Notez qu'il faut conserver tous les caractères d'espacement du cryptotexte, même s'ils se trouvent au début ou à la fin de la ligne. Il est clair que cette méthode de cryptage peut être compromise très facilement. La manière la plus simple consiste simplement à tester toutes les 26 clés possibles jusqu'à l'obtention d'un texte en clair en français.

CHIFFRE DE VIGENÈRE

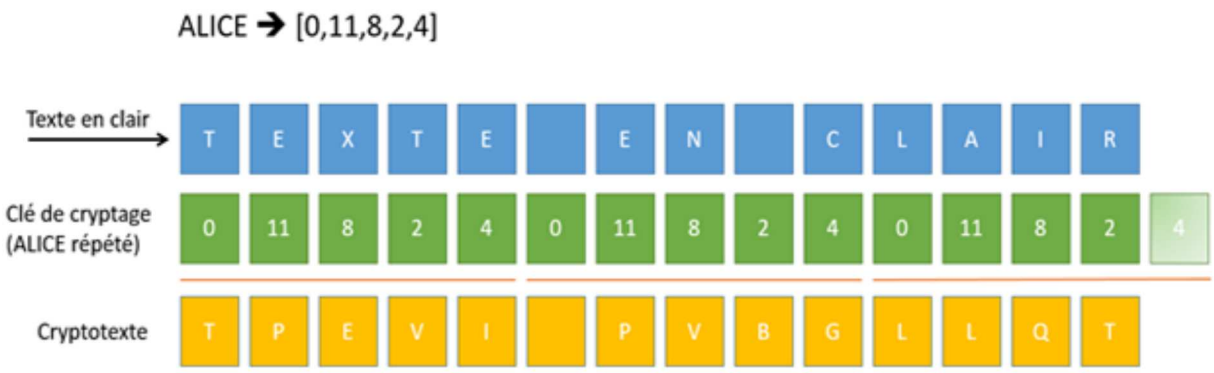
Il est possible de renforcer le chiffre de César en appliquant un décalage alphabétique différent pour chacun des caractères du texte en clair. Cette substitution poly-alphabétique peut utiliser comme clé n'importe quelle permutation de 27 nombres. Il existe donc un nombre phénoménal de clés possibles, à savoir

$27! = 10'888'869'450'418'352'160'768'000'000 \approx 10^{27}$

Il est cependant un peu plus aisé d'utiliser un mot secret auquel on fait correspondre une liste de nombres correspondant à la position de chacun de ses caractères dans l'alphabet. Ainsi, le mot secret ALICE correspond à la liste [0, 11, 8, 2, 4]. La clé de cryptage, de même longueur que le message à crypter, est construite à partir d'une juxtaposition répétée des décalages [0, 11, 8, 2, 4] correspondant au mot secret ALICE. Lors de l'encodage, les caractères du texte en clair sont décalés selon le nombre correspondant de la clé de cryptage

Blaise Vigenère (1523-1596)

L'illustration suivante permet de mieux comprendre le fonctionnement de la méthode de Vigenère:




```
import string
key = "ALICE"
alphabet = string.ascii_uppercase + " "

def encode(text):
    keyList = []
```

```
    for ch in key:
        i = alphabet.index(ch)
        keyList.append(i)
    print "keyList:", keyList
    enc = ""
    for n in range(len(text)):
        ch = text[n]
        if ch != "\n":
            i = alphabet.index(ch)
            k = n % len(key)
            ch = alphabet[(i + keyList[k]) % 27]
        enc += ch
    return enc

fInp = open("original.txt")
text = fInp.read()
fInp.close()

print "Original:\n", text
krypto = encode(text)
print "Krypto:\n", krypto

fOut = open("secret.txt", "w")
for ch in krypto:
    fOut.write(ch)
fOut.close()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

Le décodeur est à nouveau pratiquement identique à l'encodeur excepté le sens de décalage.

```
import string
key = "ALICE"
alphabet = string.ascii_uppercase + " "

def decode(text):
    keyList = []
    for ch in key:
        i = alphabet.index(ch)
        keyList.append(i)
    print "keyList:", keyList
    enc = ""
    for n in range(len(text)):
        ch = text[n]
        if ch != "\n":
            i = alphabet.index(ch)
            k = n % len(key)
            ch = alphabet[(i - keyList[k]) % 27]
        enc += ch
    return enc

fInp = open("secret.txt")
krypto = fInp.read()
fInp.close()

print "Krypto:\n", krypto
msg = decode(krypto)
print "Message:\n", msg

fOut = open("message.txt", "w")
for ch in msg:
    fOut.write(ch)
fOut.close()
```

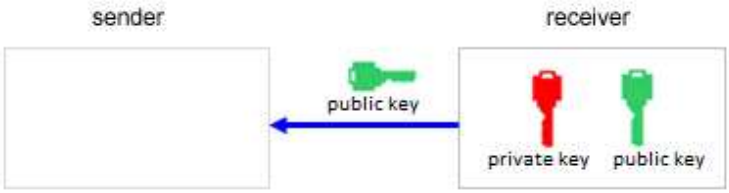
Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Le chiffre de Vigenère fut inventé au 16e siècle par Blaise de Vigenère et fut considéré comme très sûr pendant de nombreux siècles. Si quelqu'un entre en possession du cryptotexte et sait que la longueur du mot secret est 5, il lui faut néanmoins essayer systématiquement 26^5 = 11'881'376 clés différentes à moins qu'il connaisse une information supplémentaire au sujet du mot secret, comme le fait qu'il s'agit d'un prénom féminin.

CRYPTER À L'AIDE DE LA MÉTHODE RSA

Dans cette méthode dont le nom provient de celui de ses inventeurs, Rivest, Shamir et Adleman, on utilise une paire de clés, à savoir une clé privée et une clé publique. Il s'agit donc d'une méthode de cryptographie asymétrique.



Étape 1:
Le destinataire génère la *clé privée* ainsi que la *clé publique* et envoie cette dernière à l'expéditeur.

Étape 2:
L'émetteur encode son message grâce à la *clé publique* et envoie le texte crypté au destinataire.

Étape 3:
Le destinataire décode le message en utilisant sa *clé privée*.

Les clés publiques et privées sont générées en utilisant l'algorithme suivant basé sur des résultats de la théorie des nombres [**plus...**].

On commence par choisir deux nombres premiers *p* et *q* qui doivent comporter un très grand nombre de chiffres pour garantir la sécurité du cryptage. On multiplie ensuite ces deux nombres pour obtenir $m = p*q$. On sait de la théorie des nombres que l'indicatrice d'Euler $\varphi(m) = (p-1)*(q-1)$ correspond au nombre d'entiers *n* inférieurs à *m* qui sont copremiers avec *m* (*a* et *b* sont dits premiers entre eux ou copremiers si et seulement si $pgcd(m, n) = 1$).

On choisit ensuite un nombre *e* inférieur à $\varphi(m)$ tel que *e* et $\varphi(m)$ sont premiers entre eux. La paire de nombres (m, e) constitue déjà la clé publique:

Clé publique: [m, e]

Voici un exemple de calcul de la clé publique pour les petits nombres premiers p = 73 et q = 15 :

$m = 73 * 151 = 11023$, $\varphi = 72 * 150 = 10800$, $e = 11$ (choisi copremier avec φ)

Clé publique: [m, e] = [11023, 11]

La clé privée est quant à elle formée de la paire de nombres:

Clé privée: [m, d]

Où *d* est en entier tel que $(d * e) \bmod \varphi = 1$.

(Puisque *e* et φ(m)sont premiers entre eux, l'identité de Bézout affirme que cette équation possède nécessairement au moins une solution).

On peut déterminer le nombre *d* à partir des valeurs de *e* et φà l'aide d'un simple programme qui va tester 100'000 valeurs pour *d* au sein d'une boucle:

```
e = 11
phi = 10800

for d in range(100000):
    if  (d * e) % phi == 1:
        print "d", d
```

On obtient ainsi plusieurs solutions : (5891, 16691, 27491, 49091, etc.). Comme il suffit d'une seule valeur pour déterminer la clé privée, la première rencontrée fait l'affaire.

Clé privée: [m, d] = [11023, 5891]

Dans le cas présent, il est très facile de déterminer la clé privée uniquement à cause du fait que l'on connaît les nombres premiers *p* et *q* et, de ce fait, la valeur de φ. Cependant, sans la connaissance de ces nombres, il faut une puissance de calcul énorme pour calculer la clé privée.

L'algorithme RSA est utilisé pour encoder des nombres. Pour encoder du texte, on utilise donc le code ASCII de chacun des caractères du message en clair qui sera crypté à l'aide de la clé publique [m, s] selon la formule

$s = r^e$ (modulo m).

Le programme suivant écrit chacun des caractères encodés sur une nouvelle ligne dans le fichier *secret.txt*.

```
publicKey = [11023, 11]

def encode(text):
    m = publicKey[0]
    e = publicKey[1]
    enc = ""
    for ch in text:
        r = ord(ch)
        s = int(r**e % m)
        enc += str(s) + "\n"
    return enc

fInp = open("original.txt")
text = fInp.read()
fInp.close()

print "Original:\n", text
krypto = encode(text)
print "Krypto:\n", krypto

fOut = open("secret.txt", "w")
for ch in krypto:
    fOut.write(ch)
fOut.close()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

Le décodeur commence par charger ligne à ligne les nombres présents dans le fichier *secret.txt* pour les stocker dans une liste. Pour chacun des nombres présents dans cette liste, le nombre original est calculé à l'aide de la clé privée *s* selon la formule

$r = s^d$ (modulo m).

Ce nombre correspond au code ASCII du caractère original.

```
privateKey = [11023, 5891]

def decode(li):
    m = privateKey[0]
    d = privateKey[1]
    enc = ""
    for c in li:
        s = int(c)
        r = s**d % m
        enc += chr(r)
    return enc

fInp = open("secret.txt")
krypto = []
while True:
   line = fInp.readline().rstrip("\n")
   if line == "":
       break
   krypto.append(line)
fInp.close()

print "Krypto:\n", krypto
msg = decode(krypto)
print "Message:\n", msg

fOut = open("message.txt", "w")
for ch in msg:
    fOut.write(ch)
fOut.close()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Le gros avantage du code RSA réside dans le fait que l'émetteur et le récepteur n'ont pas besoin de procéder à un échange d'information secrète avant de procéder au cryptage. Au lieu de cela, le destinataire génère la clé privée et la clé publique et ne transmet que la clé publique à l'émetteur tout en gardant bien au chaud sa clé privée. L'émetteur va alors crypter son message à l'aide de la clé publique du destinataire en sachant que seul le destinataire sera capable de décrypter le message puisqu'il est le seul à disposer de la clé privée nécessaire à cette opération [**plus...**].

En pratique, on choisit de très gros nombres premiers *p* et *q* comportant des centaines de chiffres. Générer la clé publique ne nécessite que le produit m = p * q, ce qui est une banalité pour l'ordinateur. Si un pirate veut déterminer la clé privée à partir de la clé publique, il lui faut déterminer les nombres premiers originaux *p* et *q* à partir de *m*, ce qui revient à factoriser le nombre *m* en ses deux facteurs premiers. Or, la factorisation de très grands nombres premiers n'est à ce jour possible qu'en utilisant une puissance de calcul absolument colossale. On voit de ce fait que le cryptosystème RSA repose sur les limites de la calculabilité.

Il ne faut cependant jamais oublier qu'il n'existe en principe aucune méthode de cryptage parfaitement incassable. Heureusement, il suffit que l'opération de décryptage soit considérablement plus longue que la période de validité ou de pertinence de l'information pour que la méthode soit considérée comme étant suffisamment sûre.

EXERCICES

1. Décoder le message crypté suivant:
   OAL SHDXTKMJXSASTVVXHLSL XSAFNALTLAGF
   UMLSASVTFSGFDQSUXSL XJXSTLSXAZ L
   ZJXXLAFZK
   XNXDAFX

   Il s'agit d'un chiffre de César.

   Remarque : Une solution efficace repose sur le fait que la lettre la plus fréquente en français est la lettre E. Il est cependant également possible de procéder en testant tous les décalages possibles.

2. Faire une recherche sur le Web concernant le système de cryptage Skytale et implémenter un encodeur / décodeur basé sur ce principe.

3. Expliquer en quoi le chiffre de César est un cas particulier de chiffre de Vigenère.

4. Utiliser la méthode RSA pour générer une clé publique et une clé privée à l'aide de deux nombres premiers p et q tous deux inférieurs à 100 et procéder au cryptage / décryptage d'un message de votre choix

10.6 AUTOMATES FINIS

INTRODUCTION

Il est nécessaire de définir exactement ce qu'est un ordinateur avant de pouvoir déterminer dans quelle mesure il est capable de résoudre tel ou tel problème. Le célèbre mathématicien et informaticien Alan Turing publia une étude à ce sujet en 1936, bien avant que le premier ordinateur digital programmable n'existe. La machine de Turing, baptisée ainsi en l'honneur de Turing, passe successivement par différents états de manière programmée sur la base de données lues sur une bande et écrites sur cette même bande. Cette notion fondamentale qui touche au fonctionnement de l'ordinateur est encore valable actuellement puisque tous les processeurs présents dans nos ordinateurs sont en fait des machines de Turing qui passent d'un état à l'autre à la cadence de l'horloge. Cependant, les machines à états qui peuvent être modélisées par un graphe de transition se prêtent mieux aux application pratiques. Du fait qu'elles ne disposent que d'un nombre fini d'états, on les appelle des **automates finis** (*finite-state machines* en anglais).

PROGRAMMING CONCEPTS: *Machine de Turing, machines (automates) finis, machine de Mealy, graphe de transition, théorie des langages formels*

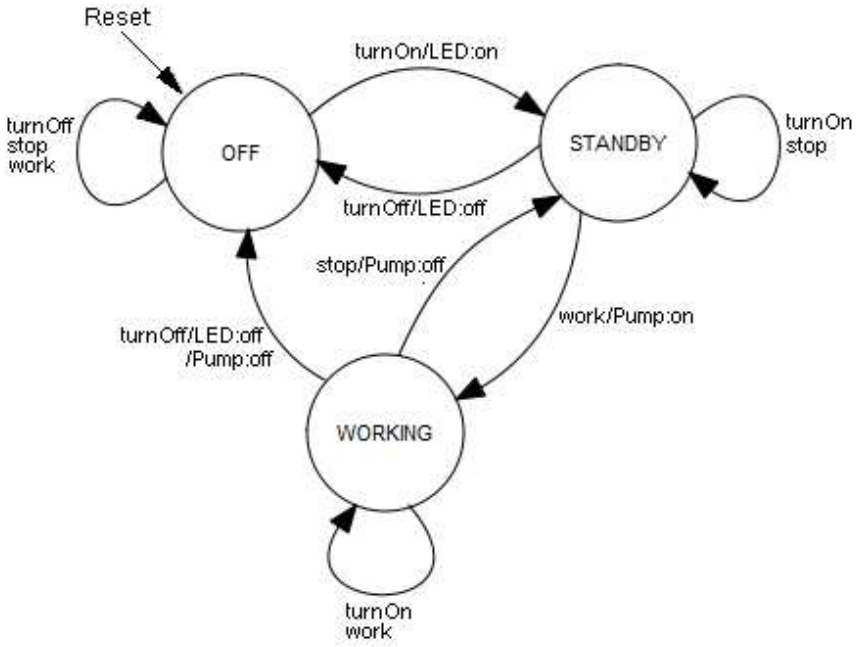
LA MACHINE ESPRESSO COMME MACHINE DE MEALY

Chaque jour, nous sommes tous confrontés à des appareils et machines qui peuvent être considérés comme des automates parmi lesquels on trouve les distributeurs automatiques de boissons ou de billets de banque, les machines à laver et tant d'autres. Un ingénieur ou un informaticien développant de telles machines se doit de comprendre clairement qu'elles effectuent leur tâche en passant de l'état courant à un état successeur. Ces transitions d'état sont déterminées par **les entrées** de l'automate, à savoir les données en provenance des capteurs ou des interrupteurs formant l'interface de commande. En fonction des données en entrée, l'opérateur actionne certains actuateurs lors de chaque transition d'états tels que des moteurs, des pompes, des témoins lumineux qui constituent **les sorties** de l'automate.

Dans le cas présent, nous allons développer une machine à café espresso pouvant se trouver dans trois états différents : elle peut être éteinte (OFF), allumée et en attente (STANDBY) ou en train de pomper et chauffer de l'eau pour préparer un espresso (WORKING). L'interface de contrôle de la machine comporte quatre boutons-poussoirs : *turnOn*, *turnOff*, *work*, et *stop*.

Bien qu'il soit possible de décrire le fonctionnement d'une machine à café espresso en prose, un graphe de transition est bien plus clair et explicite. Un tel graphe est formé de cercles représentant les différents états (nœuds du graphe) et de flèches (arête orientée) passant d'un état à l'autre et formalisant les transitions d'états. Les flèches sont étiquetées par les entrées / sorties qui causent la transition en question. Il est également important de définir l'état initial de la machine lorsqu'elle est mise en fonction. Du fait que n'importe quelle touche peut être actionnée dans n'importe lequel des états, toutes les entrées doivent être envisagées à chaque état. Si aucune action n'est effectuée, la sortie est omise.

Graphe de transition :



On résume souvent le comportement de la machine dans un tableau qui spécifie pour chaque état *s* et chaque entrée *t* (ici les boutons actionnés) l'état successeur *s'*. L'état initial est dénoté par une étoile.

Table de transition:

t = s =	**OFF(*)**	**STANDBY**	**WORKING**
turnOff	OFF	OFF	OFF
turnOn	STANDBY	STANDBY	WORKING
stop	OFF	STANDBY	STANDBY
work	OFF	STANDBY	WORKING

En termes mathématiques, on peut dire que l'état successeur *s'* est une fonction de l'état courant et de l'entrée *t* : *s' = F(s, t)*. La fonction F est appelée **fonction de transition**.

On peut également spécifier dans un tel tableau les sorties correspondant à chaque état et chaque donnée en entrée :

Table de sortie:

t = s =	**OFF(*)**	**STANDBY**	**WORKING**
turnOff	-	LED éteinte	LED éteinte, Pompe allumé
turnOn	LED allumée	-	-
stop	-	-	Pump éteinte
work	-	Pump allumée	-

À nouveau, on peut dire que, mathématiquement, la sortie *g* est une fonction de l'état courant et des entrées : *g = G(s, t). G est appelée* ***fonction de sortie****.*

MEMENTO

Tout ceci, à savoir les différents états, les valeurs en entrée, les valeurs en sortie ainsi que les fonctions de transition et de sortie forment une **machine de Mealy**.

IMPLÉMENTATION DE LA MACHINE ESPRESSO AVEC DES CHAÎNES DE CARACTÈRES

La pression d'un bouton devrait engendrer une transition d'un état à l'état suivant. Les 4 touches directionnelles du clavier serviront à simuler les boutons de la machine et spécifient les valeurs en entrée. L'implémentation est relativement triviale : le programme attend qu'une touche soit enfoncée au sein d'une boucle d'événements (*event loop*) infinie avec *getKeyEvent()*. Selon la valeur de retour de *getKeyEvent()*, l'état courant est modifié d'après la table de transition et les sorties sont calculées en fonction de la table de sortie.

```
from gconsole import *

def getKeyEvent():
    keyCode = getKeyCodeWait(True)
    if keyCode == KeyEvent.VK_UP:
        return "stop"
    if keyCode == KeyEvent.VK_DOWN:
        return "work"
    if keyCode == KeyEvent.VK_LEFT:
        return "turnOff"
    if keyCode == KeyEvent.VK_RIGHT:
        return "turnOn"
    return ""

state = "OFF"  # Start state
makeConsole()
while True:
    gprintln("State: " + state)
    entry = getKeyEvent()
    if entry == "turnOff":
       if state == "STANDBY":
            state = "OFF"
            gprintln("LED off")
       if state == "WORKING":
            state = "OFF"
            gprintln("LED and pump off")
    elif entry == "turnOn":
        if state == "OFF":
            state = "STANDBY"
            gprintln("LED enabled")
    elif entry == "stop":
        if state == "WORKING":
            state = "STANDBY"
            gprintln("Pumpe off")
    elif entry == "work":
        if state == "STANDBY":
            state = "WORKING"
            gprintln("Pumpe enabled")
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Seuls les événements qui mènent à un changement d'état ou qui génèrent une sortie sont traités au sein de la boucle d'événements.

La fonction *makeConsole()* permet de créer une fenêtre d'entrée / sortie très simple qui accepte des saisies de caractères individuels au clavier sans la nécessité de devoir appuyer sur <return> pour être validés. La fonction *getKeyCodeWait()* met le programme en attente de la pression d'une touche du clavier et retourne le code de la touche pressée. La documentation du module *gconsole* se trouve dans l'aide de TigerJython sous *Aide / APLU documentation*.

TYPES ÉNUMÉRÉS COMME ÉTATS ET IDENTIFIANTS D'ÉVÉNEMENTS

Puisque les automates fonctionnent sur la base d'états, de valeurs en entrée et de valeurs en sortie, il serait avantageux d'introduire une structure de données particulière spécialement adaptée. De nombreux langages de programmation mettent à disposition un type de données particulier pour les énumérations. Malheureusement, ce type de données n'est pas présent dans la syntaxe du langage Python standard. Heureusement, il a été ajouté à TigerJython grâce au mot-clé additionnel *enum()* qui permet de définir des valeurs énumérées à partir de chaines de caractères respectant les contraintes et conventions de nommage en vigueur pour les noms de variables.

```
from gconsole import *

def getKeyEvent():
    keyCode = getKeyCodeWait(True)
    if keyCode == KeyEvent.VK_UP:
        return Events.stop
    if keyCode == KeyEvent.VK_DOWN:
        return Events.work
    if keyCode == KeyEvent.VK_LEFT:
        return Events.turnOff
    if keyCode == KeyEvent.VK_RIGHT:
        return Events.turnOn
    return None

State = enum("OFF", "STANDBY", "WORKING")
state = State.OFF
Events = enum("turnOn", "turnOff", "stop", "work")
makeConsole()
while True:
    gprintln("State: " + str(state))
    entry = getKeyEvent()
    if entry == Events.turnOn:
        if state == State.OFF:
            state = State.STANDBY
    elif entry == Events.turnOff:
            state = State.OFF
    elif entry == Events.work:
        if state == State.STANDBY:
            state = State.WORKING
    elif entry == Events.stop:
        if state == State.WORKING:
            state = State.STANDBY
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Il est de votre ressort de décider si vous voulez ou non utiliser le type de données additionnel *enum*. Son usage ne rend pas les programmes plus courts mais plus lisibles et sécurisés du fait que seules les valeurs explicitement spécifiées dans le type énuméré seront reconnues et autorisées.

IMPLÉMENTATION AVEC CONTRÔLE DE LA SOURIS

Il suffit d'un petit effort supplémentaire pour mettre sur pied une simulation graphique de la machine Espresso en utilisant la bibliothèque *JGameGrid*, ce qui ne manquera pas de rendre la simulation encore plus claire et amusante. En lieu et place des quatre touches directionnelles du clavier, les quatre entrées sont actionnées par des clics de souris sur des boutons simulés et la sortie de la LED et du pompage sont immédiatement rendus visibles par des images de sprite. Au lieu d'utiliser une boucle d'événements, il est fait usage de la fonction de rappel *pressEvent()* qui sera toujours invoquée lors d'un clic de souris sur l'image de la machine avec la souris. Puisque l'on utilise une grille comportant 7 x 11 cellules en tant que *GameGrid*, il est possible de capturer les clics effectués sur les boutons virtuels à l'aide des coordonnées grille.

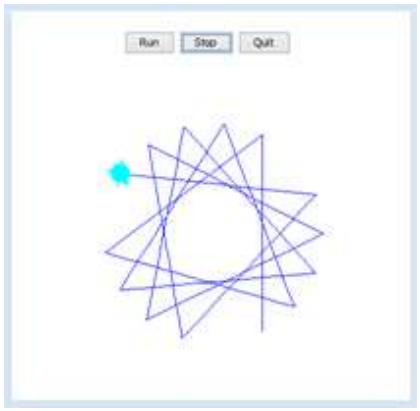
```
from gamegrid import *

def pressEvent(e):
    global state
    loc = toLocationInGrid(e.getX(), e.getY())
    if loc == Location(1, 2): # off
        state = State.OFF
        led.show(0)
        coffee.hide()
    elif loc == Location(2, 2): # on
        if state == State.OFF:
            state = State.STANDBY
            led.show(1)
    elif loc == Location(4, 2): # stop
        if state == State.WORKING:
            state = State.STANDBY
            coffee.hide()
    elif loc == Location(5, 2): # work
        if state == State.STANDBY:
            state = State.WORKING
            coffee.show()
    setTitle("State: " + str(state))
    refresh()

State = enum("OFF", "STANDBY", "WORKING")
state = State.OFF
makeGameGrid(7, 11, 50, None, "sprites/espresso.png", False,
             mousePressed = pressEvent)
show()
setTitle("State: " + str(state))
led = Actor("sprites/lightout.gif", 2)
addActor(led, Location(3, 3))
coffee = Actor("sprites/coffee.png")
addActor(coffee, Location(3, 6))
coffee.hide()
refresh()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

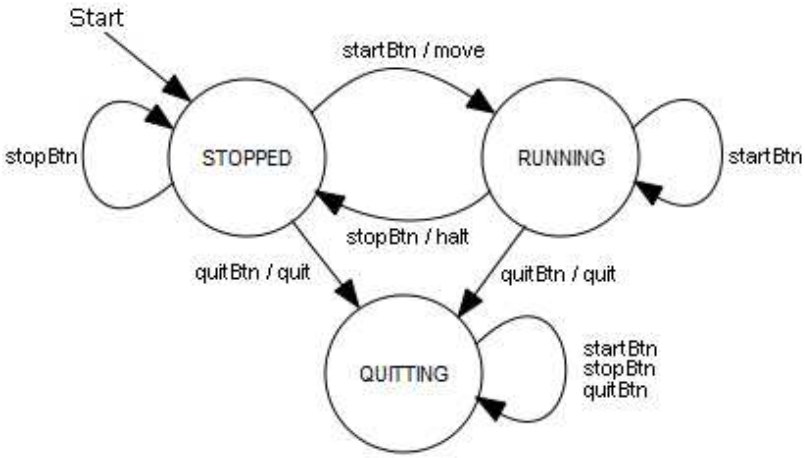
MEMENTO

Une interface utilisateur graphique permet de rendre une simulation beaucoup plus claire et attractive.

PENSER LES INTERFACES GRAPHIQUES EN TERMES D'ÉTATS



À première vue, les machines de Mealy semblent être confinées au monde des théories fumeuses. Mais il faut, bien au contraire, toujours penser en termes d'états lors du développement de programmes événementiels comportant une interface utilisateur graphique. En guise d'exemple, écrivons un petit programme tortue contrôlé par 3 boutons. Le bouton *start_button* démarre le mouvement de la tortue, le bouton *stop_button* l'interrompt et le bouton *quit_button* termine le programme. Afin d'implémenter ce programme correctement, il est nécessaire de garder en tête le graphe de transition:



Comme nous l'avons déjà vu à maintes reprises, il ne faut jamais effectuer le rendu d'une animation au sein des gestionnaires d'événements de l'interface graphique car ces derniers ne supportent que l'exécution d'un code très rapide. Cela vient du fait que le rendu à l'écran n'est effectué qu'à la fin de la fonction de rappel. C'est la raison pour laquelle on se contente, au sein du gestionnaire de clic sur les boutons, de modifier l'état de sorte que le mouvement de la tortue est effectué dans la partie principale du programme.

(Vous pouvez en apprendre davantage au sujet de ce problème dans l'annexe 4 **Traitement parallèleg**)

```
from javax.swing import JButton
from gturtle import *

def buttonCallback(evt):
    global state
    source = evt.getSource()
    if source == runBtn:
        state = State.RUNNING
        setTitle("State: RUNNING")
    if source == stopBtn:
        state = State.STOPPED
        setTitle("State: STOPPED")
    if source == quitBtn:
        state = State.QUITTING
        setTitle("State: QUITTING")

State = enum("STOPPED", "RUNNING", "QUITTING")
state = State.STOPPED

runBtn = JButton("Run", actionPerformed = buttonCallback)
```

```
stopBtn = JButton("Stop", actionPerformed = buttonCallback)
quitBtn = JButton("Quit", actionPerformed = buttonCallback)
makeTurtle()
setTitle("State: STOPPED")
back(100)

pg = getPlayground()
pg.add(runBtn)
pg.add(stopBtn)
pg.add(quitBtn)
pg.validate()

while state != State.QUITTING and not isDisposed():
    if state == State.RUNNING:
          forward(200).left(127)
dispose()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

La structure de ce code est typique de la programmation événementielle. Vous devriez donc faire votre maximum pour la mémoriser tant elle est fréquente.

Il est nécessaire d'importer le module *JButton* pour utiliser les boutons qu'il faut ajouter à la fenêtre de tortue (objet *Playground*) à l'aide de sa méthode *add()*. Il faut ensuite rafraîchir la fenêtre tortue avec la méthode *validate()* pour que ces boutons soient visibles.

EXERCICES

1. Un parcomètre n'accepte que des pièces de 1 € et 2 € qui sont à insérer l'une après l'autre. Dès que la machine a reçu suffisamment d'argent pour payer le parking, elle émet le billet et retourne la monnaie au cas où le montant inséré s'avère trop important. La taxe de parking se monte à 3 €.

   En partant de l'état initial S0, la machine passe aux état S1 ou S2, selon qu'une pièce de 1 € ou de 2 €a été insérée. Elle imprime sur la sortie les valeurs « - » (rien), « K » (ticket) ou « K,R » (ticket et monnaie de retour).

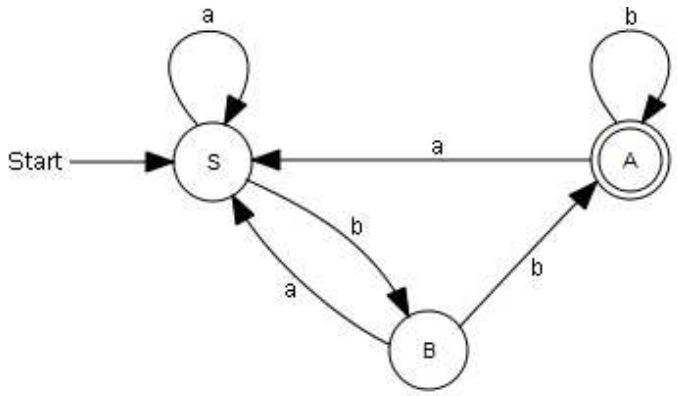
   a. Déterminer les tables d'entrées et de sorties de cet automate
   b. Dessiner le graphe de transition du parcomètre
   c. Développer, à l'aide de *GConsole (voir premier exemple ci-dessus)*, un programme qui interprète la pression de la touche 1 du clavier comme l'insertion d'une pièce de 1 € et de la touche 2 comme l'insertion d'une pièce de 2 €. Le programme écrira l'état subséquent ainsi que les valeurs de sortie dans la console.

MATÉRIEL SUPPLÉMENTAIRE

ACCEPTEURS POUR LANGAGES RATIONNELS (RÉGULIERS)

Un langage formel est constitué d'un alphabet de symboles et d'un jeu de règles permettant de déterminer de manière univoque si une séquence de symboles particulière appartient ou non au langage en question. S'il est possible d'implémenter les règles en utilisant un automate, on parle de **langage régulier** ou **langage rationnel**.

Comme exemple, considérons un langage très simple dont l'alphabet est constitué uniquement des lettres A et B. On peut considérer le jeu de règles comme une machine de Mealy spéciale qui n'engendre aucune valeur de sortie. En l'occurrence, la machine lit l'expression caractère après caractère en partant d'un état initial et effectue une transition vers l'état successeur sur la base du caractère lu. Si l'automate se trouve dans l'un des états finaux prédéterminés après la lecture de la séquence de caractères, l'expression en question appartient au langage. Considérons le graphe de transition suivant (S: état initial, A: état final):



Dans l'implémentation ci-dessous, le changement d'état est déclenché par la pression de la touche A ou B du clavier. Le programme imprime ensuite l'état courant et le mot saisi dans la console.

```
from gconsole import *

def getKeyEvent():
    global word
    keyCode = getKeyCodeWait(True)
    if keyCode == KeyEvent.VK_A:
        return Events.a
    if keyCode == KeyEvent.VK_B:
        return Events.b
    return None

State = enum("S", "A", "B")
state = State.S
Events = enum("a", "b")
makeConsole()
word = ""
gprintln("State: " + str(state))
while True:
    entry = getKeyEvent()
    if entry == Events.a:
        if state == State.A:
            state = State.S
        elif state == State.B:
            state = State.S
        word += "a"
        gprint("Word: " + word + " -> ")
        gprintln("State: " + str(state))
    elif entry == Events.b:
        if state == State.S:
            state = State.B
```

```
        elif state == State.B:
            state = State.A
        word += "b"
        gprint("Word: " + word + " -> ")
        gprintln("State: " + str(state))
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Un accepteur permet de déterminer si un mot donné appartient ou non au langage. Il s'agit d'un cas particulier de machine de Mealy dépourvue de valeur de sortie. Le mot appartient au langage si la lecture de ses caractères individuels permet de passer de l'état initial à un des états finaux acceptables. Le mot *abbabb* appartient par exemple au langage alors que ce n'est pas le cas de *baabaa*.

EXERCICES

1. Une machine à rire ne doit accepter que les mots *ha.* ou *haha.* ou encore *hahaha.* etc. Le dernier caractère de la séquence est un point (.). Dessiner le graphe de transition de cette machine. Implémenter la machine correspondante en Python.

   Instructions supplémentaires : introduire un état d'erreur E qu'il n'est plus possible de quitter quelle que soit l'entrée.

10.7 INFORMATION ET ORDRE

INTRODUCTION

Bien que le mot *information* soit utilisé très couramment, il n'est pas si facile de saisir avec précision le concept d'information en tant que grandeur mesurable. Dans la vie courante, l'information est liée à la connaissance et dire que quelqu'un dispose d'avantage d'information signifie que cette personne en sait plus sur un certain sujet qu'une personne B qui n'est pas bien informée.

Pour quantifier le surplus d'information dont dispose A par rapport à B (ou à l'équipe B) ou, à l'inverse, le manque d'information de B par rapport à A, il faut imaginer un jeu de quiz télévisé. La personne ou l'équipe B doit découvrir une information que seule la personne A connaît comme par exemple sa profession. Pour ce faire, B peut poser des questions auxquelles A ne répondra que par « oui » ou « non ». Les règles sont les suivantes:

Le défaut d'information I de B par rapport à A (en bits) est le nombre de questions fermées (réponse oui/non) que B doit poser à A en moyenne et avec une stratégie optimale pour disposer des mêmes connaissances que A sur le sujet en question.

CONCEPTS DE PROGRAMMATION: *Information, quantité d'information, entropie*

DEVINETTES DE NOMBRES

Vous pourriez facilement imaginer un tel jeu dans votre classe ou juste dans votre esprit de la manière suivante : votre camarade Léa sort de la salle et l'un de ses W = 16 camarades de classe reçoit un objet désirable tel qu'une barre de chocolat. Lorsque Léa réintègre la salle, vous et vos camarades savez qui détient la barre de chocolat alors que Léa n'en sait rien. Comment quantifier ce manque d'information dont souffre Léa?

Pour simplifier les explications, numérotons les élèves de 0 à 15. Léa peut alors poser n'importe quelle question à ses camarades dans le but de trouver le nombre secret qui lui vaudra la barre de chocolat. Son but est de poser le moins de questions possible jusqu'à pouvoir trouver de manière certaine la personne cachant la branche. Elle aurait par exemple la possibilité d'essayer successivement n'importe quel nombre aléatoirement : « est-ce le 13 ? ». Si la réponse est « non », elle reprend en demandant un autre nombre au bol.

Léa pourrait également procéder de manière systématique et essayer tous les nombres de 0 à 15. Dans la simulation informatique ci-dessous, on se demande combien elle devrait poser de questions en moyenne (à savoir si le jeu est rejoué de nombreuses fois) pour tomber sur la barre de chocolat avec cette stratégie de questionnement.

Si le nombre secret est « 0 », il suffit alors d'une seule question. Si le nombre secret est « 1 », deux questions seront nécessaires et ainsi de suite. Si le nombre secret est « 14 », il faut alors 15 questions de même que si le nombre secret est 15 puisqu'il ne reste alors plus qu'une seule alternative. Le programme simule le jeu 100'000 fois et compte le nombre de questions nécessaires jusqu'à ce que le nombre secret soit découvert. Ces résultats sont additionnés pour en déterminer la moyenne.

```
import random
```

```
sum = 0
z = 100000
repeat z:
    n = random.randint(0, 15)
    if n != 15:
        q = n + 1 # number of questions
    else:
       q = 15
    sum += q
print "Mean:", sum / z
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

En utilisant cette stratégie, Léa devrait poser en moyenne 8.43 questions, ce qui est tout de même assez important. Mais comme Léa est intelligente, elle décide d'utiliser une stratégie bien plus efficace. Elle commence par demande : « Est-ce un nombre entre 0 et 7, bornes incluses ? ». Si la réponse est « oui », elle divise les possibilités restantes en deux groupes et demande : « Est-ce un nombre entre 0 et 3 ? ». Si la réponse est à nouveau « oui », elle demande alors : « Est-ce un nombre compris entre 0 et 1 (compris) ? ». Si la réponse est à nouveau positive, elle demande : « Est-ce 0 ? ». À ce stade, elle connaîtra la réponse quelle que soit la réponse. Avec cette stratégie d'interrogation « binaire » et pour W=16, Léa trouve toujours le nombre secret en exactement quatre questions, ce qui représente donc également le nombre moyen de questions à poser. Cela montre que la stratégie de bissection est optimale et que l'information concernant la personne cachant la barre est donc quantifiée à I = 4 bits. .

MEMENTO

Comme vous pouvez le montrer vous-mêmes, la quantité d'information pour W = 32 vaut I = 5 bits et vaudra 6 bits pour W = 64. Ainsi, $2^I = W$ *ou, autrement dit,* $I = ld(W)$.). L'information en question n'est pas nécessairement un nombre et peut résider dans le fait de déterminer un état quelconque parmi un nombre W d'états équiprobables.

Ainsi, l'**information** découlant de la connaissance d'un état donné parmi W états équiprobables est donnée par

$$I = ld(W)$$ (où ld est le « logaritmus dualis », à savoir le *log* en base 2)

QUANTITÉ D'INFORMATION CONTENUE DANS UN MOT

Comme les W états sont équiprobables, la probabilité de chacun des états vaut p = 1/W, ce qui permet également d'écrire l'information de la manière suivante

$$I = ld(1/p) = -ld(p)$$

Vous vous doutez bien que les probabilités d'occurrence de chacune des lettres de l'alphabet sont sensiblement différentes d'une langue à l'autre. Le tableau suivant montre la probabilité d'occurrence de chacune des lettres en anglais [**plus...**].

A	8.34%	I	6.71%	Q	0.09%	Y	2.04%
B	1.54%	J	0.23%	R	5.68%	Z	0.96%
C	2.73%	K	0.87%	S	6.11%		
D	4.14%	L	4.24%	T	9.37%		
E	12.60%	M	2.53%	U	2.85%		
F	2.03%	N	6.80%	V	1.06%		
G	1.92%	O	7.70%	W	2.34%		
H	6.11%	P	1.66%	X	0.20%		

Pour simplifier, supposons que la probabilité d'occurrence d'une lettre dans un mot soit indépendante des autres caractères. Bien que cette hypothèse ne soit certainement pas réaliste, elle permet de conclure que la probabilité *p* d'un mot de deux lettres de probabilités respectives *p1* et *p2* est, d'après la formule du produit, $p = p1 * p2$ et, concernant l'information:

$$I = -ld(p) = -ld(p_1 * p_2) = -ld(p_1) - ld(p_2)$$

ou, pour un nombre quelconque de lettres,

$$I = - ld(p_1) - ld(p_2) - ld(p_3) - ... - ld(p_n) = - \Sigma\, ld(p_i)$$

Le programme suivant permet de saisir un mot et d'afficher en sortie la quantité d'information contenue dans ce mot. Pour cela, il est nécessaire de télécharger les fichiers comportant la fréquence pour l'allemand, l'anglais et le français grâce à **ce lien** et de les copier dans le dossier où réside votre script Python.

```
import math

f = open("letterfreq_de.txt")
s = "{"
for line in f:
    line = line[:-1] # remove trailing \n
    s += line + ", "
f.close()
s = s[:-2]  # remove trailing ,
s += "}"
occurance = eval(s)

while True:
    word = inputString("Enter a word")
    I = 0
    for letter in word.upper():
        p = occurance[letter] / 100
        I -= math.log(p, 2)
    print word, "-> I =", round(I, 2), "bit"
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Les données sont stockées dans le fichier en question ligne à ligne au format *'lettre' : pourcentage*. La structure de données Python qui se prête le mieux à représenter ces données est le dictionnaire. Le fichier est donc lu ligne à ligne et empaqueté dans une chaine de caractères au format approprié {clé : valeur, clé : valeur, ...}. Cette chaine de caractères est ensuite évaluée par l'interpréteur Python comme du Python grâce à la fonction *eval()* et le dictionnaire est créé.

Comme vous pouvez le constater, la quantité d'information contenue dans les mots comprenant des lettres peu fréquentes est supérieure. Il faut bien noter que la quantité d'information déterminée de cette manière n'a rien à voir avec l'importance ou la pertinence subjective du mot dans un contexte donné. Autrement dit, cette mesure ne permet pas de savoir si l'information véhiculée par ce mot vous laisse complètement indifférent ou si, au contraire, elle est pour vous d'une importance cruciale. La détermination d'une telle mesure dépasse d'ailleurs largement la capacité des systèmes d'information actuels.

EXERCICES

1. Lors du jeu de devinettes avec ses 16 camarades, Léa aurait également pu procéder d'une manière tout aussi efficace en demandant à ses camarades de se représenter le nombre secret en représentation binaire de 4 chiffres. Décrire une manière optimale de procéder pour trouver le nombre secret en se basant sur sa représentation binaire.

2. Utiliser la liste de mots anglais contenus dans le fichier *words-1$.txt* pour déterminer le mot contenant la plus petite quantité d'information et celui en comportant le plus parmi tous les mots de 5 lettres. Commenter vos résultats. La liste de mots peut être téléchargée depuis **ce lien**.

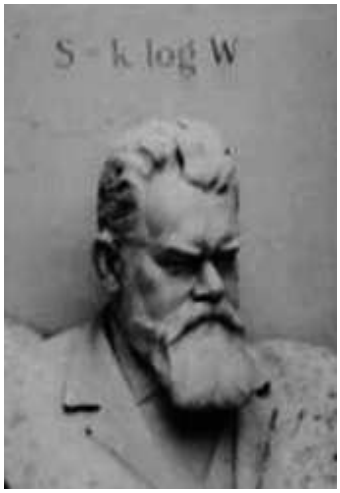
3*. Déterminer de manière théorique, en utilisant des concepts mathématiques, le nombre moyen de questions nécessaires pour déterminer le nombre secret en utilisant la méthode consistant à demander successivement tous les nombres en partant de 0 jusqu'à 15.

MATÉRIEL SUPPLÉMENTAIRE

RELATION ENTRE DÉSORDRE ET ENTROPIE

Il existe une relation très intéressante entre l'information dont on dispose à propos d'un système et son niveau d'ordre. L'ordre dans une classe dont tous les élèves sont sagement assis à leur place est certainement bien plus élevé que s'ils sont tous en train de se déplacer de manière aléatoire dans la salle. Dans le cas de l'état désordonné, il est très difficile de connaître la position de chacun des élèves. On peut donc utiliser cette mesure de notre défaut de connaissance pour avoir une idée du niveau de désordre dans la classe. Le concept qui se cache derrière cette quantification du désordre est l'**entropie**.

L'entropie d'un système (en bits) correspond au défaut d'information I d'une personne décrivant le système de manière macroscopique par rapport à une personne qui en connaît l'état microscopique. L'entropie (J/K) d'un système est donnée par la relation $S = k * ln2 * I$



Le facteur k est la constante de Bolzmann. L'entropie et le défaut de connaissance sont donc identiques à un facteur près. En supposant que le système est composé de W états équiprobables, on a

$S = k * ln2 * ldW$ où **$S = k * log\ W$**

Cette relation fondamentale provient du fameux physicien Ludwig Boltzmann (1844-1900) et se trouve gravée sur sa pierre tombale.

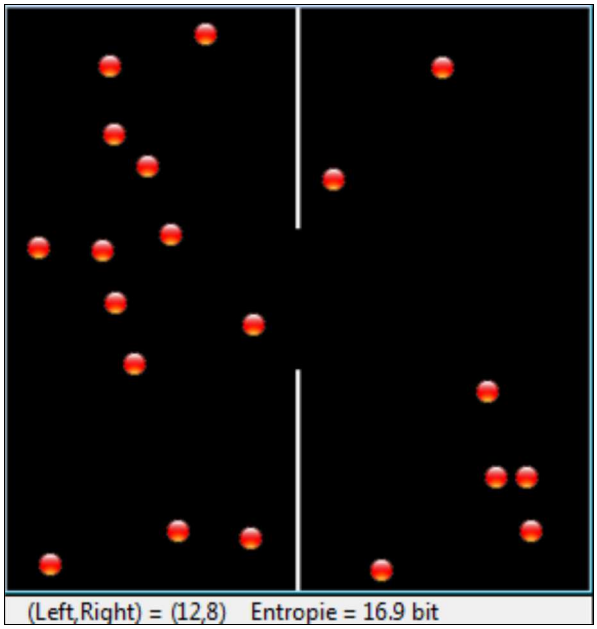
Comme le montrent l'expérience quotidienne ainsi que la simulation ci-dessous, les systèmes qui sont abandonnés à leur propre sort ont tendance à passer d'un état ordonné à un état désordonné. Voici quelques exemples concrets :

- Les passagers d'un train ont tendance à se répartir dans l'ensemble du wagon
- La fumée de cigarette se répartit dans toute la pièce
- L'encre se disperse dans un verre d'eau
- La température a tendance à s'équilibrer entre la tasse de café chaude et l'air à température ambiante.

Since the disordered state has a higher entropy than the ordered state, one can formulate this as a law of nature (2nd law of thermodynamics):

Dans un système fermé, l'entropie augmente ou reste constante mais elle ne va jamais diminuer.

Le programme ci-dessous simule un système de particules d'atomes de gaz qui entrent en collisions les unes avec les autres en échangeant la direction de leur vitesse et leur énergie cinétique. Toutes les particules se trouvent initialement dans la partie gauche du récipient dont elles peuvent s'échapper par un petit trou dans la paroi séparatrice. Que se passe-t-il ?



Les animations sont réalisées à l'aide du module *JGameGrid.* Les particules sont modélisées par la classe *Particle* dérivant de *Actor* et se déplacent grâce à leur méthode *act()* qui est appelée automatiquement lors de chaque cycle de simulation. Les collisions sont gérées par des événements en dérivant la classe *CollisionListener* de la classe *GGActorCollisionListener* et en redéfinissant la méthode *collide()*.

Ce procédé est le même que celui mis en œuvre au chapitre 8.10 dans la simulation du **mouvement browninien**. Les 20 particules sont ensuite réparties en 4 groupes de vitesses différents. Étant donné que les vitesses sont échangées entre particules, l'énergie totale du système demeure constante, ce qui correspond à un système fermé.

```
from gamegrid import *
from gpanel import *
import math
import random

# ================== class Particle ====================
class Particle(Actor):
    def __init__(self):
        Actor.__init__(self, "sprites/ball_0.gif")

    # Called when actor is added to gamegrid
    def reset(self):
        self.isLeft = True

    def advance(self, distance):
        pt = self.gameGrid.toPoint(self.getNextMoveLocation())
        dir = self.getDirection()
        # Left/right wall
        if pt.x < 0 or pt.x > w:
            self.setDirection(180 - dir)
        # Top/bottom wall
        if pt.y < 0 or pt.y > h:
            self.setDirection(360 - dir)
        # Separation
        if (pt.y < h // 2 - r or pt.y > h // 2 + r) and \
            pt.x > self.gameGrid.getPgWidth() // 2 - 2 and \
            pt.x < self.gameGrid.getPgWidth() // 2 + 2:
                self.setDirection(180 - dir)

        self.move(distance)
        if self.getX() < w // 2:
            self.isLeft = True
        else:
```

```
            self.isLeft = False

    def act(self):
        self.advance(3)

    def atLeft(self):
        return self.isLeft

# ================== class CollisionListener =========
class CollisionListener(GGActorCollisionListener):
    # Collision callback: just exchange direction and speed
    def collide(self, a,  b):
        dir1 = a.getDirection()
        dir2 = b.getDirection()
        sd1 = a.getSlowDown()
        sd2 = b.getSlowDown()
        a.setDirection(dir2)
        a.setSlowDown(sd2)
        b.setDirection(dir1)
        b.setSlowDown(sd1)
        return 5  # Wait a moment until collision is rearmed


# ================== Global sections =================
def drawSeparation():
    getBg().setLineWidth(3)
    getBg().drawLine(w // 2, 0, w // 2, h // 2 - r)
    getBg().drawLine(w // 2, h, w // 2, h // 2 + r)

def init():
    collisionListener = CollisionListener()
    for i in range(nbParticles):
        particles[i] = Particle()

        # Put them at random locations, but apart of each other
        ok = False
        while not ok:
            ok = True
            loc = getRandomLocation()
            if loc.x > w / 2 - 20:
                ok = False
            continue

        for k in range(i):
            dx = particles[k].getLocation().x - loc.x
            dy = particles[k].getLocation().y - loc.y
            if dx * dx + dy * dy < 300:
                ok = False
        addActor(particles[i], loc, getRandomDirection())
        delay(100)

        # Select collision area
        particles[i].setCollisionCircle(Point(0, 0), 8)
        # Select collision listener
        particles[i].addActorCollisionListener(collisionListener)

        # Set speed in groups of 5
        if i < 5:
            particles[i].setSlowDown(2)
        elif i < 10:
            particles[i].setSlowDown(3)
        elif i < 15:
            particles[i].setSlowDown(4)

    # Define collision partners
    for i in range(nbParticles):
        for k in range(i + 1, nbParticles):
            particles[i].addCollisionActor(particles[k])
```

```
def binomial(n, k):
    if k < 0 or k > n:
        return 0
    if k == 0 or k == n:
        return 1
    k = min(k, n - k) # take advantage of symmetry
    c = 1
    for i in range(k):
        c = c * (n - i) / (i + 1)
    return c

r = 50  # Radius of hole
w = 400
h = 400
nbParticles = 20
particles = [0] * nbParticles
makeGPanel(Size(600, 300))
window(-6, 66, -2, 22)
title("Entropy")
windowPosition(600, 20)
drawGrid(0, 60, 0, 20)
makeGameGrid(w, h, 1, False)
setSimulationPeriod(10)
addStatusBar(20)
drawSeparation()
setTitle("Entropy")
show()
init()
doRun()

t = 0
while not isDisposed():
    nbLeft = 0
    for particle in particles:
        if particle.atLeft():
            nbLeft += 1
    entropy = round(math.log(binomial(nbParticles, nbLeft), 2), 1)
    setStatusText("(Left,Right) = (" + str(nbLeft) +
           "," + str(nbParticles - nbLeft) + ")" +
           "    Entropie = " + str(entropy) + " bit")
    if t % 60 == 0:
        clear()
        lineWidth(1)
        drawGrid(0, 60, 0, 20)
        lineWidth(3)
        move(0, entropy)
    else:
        draw(t % 60, entropy)
    t += 1
    delay(1000)
dispose() # GPanel
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Vu de l'extérieur (d'un point de vue macroscopique), un état est déterminé par le nombre *k* de ses particules dans la partie droite et, de ce fait, par les *N*-*k* particules présentes dans sa partie gauche. Une observation macroscopique ignore complètement lesquelles des *k* particules parmi les *N* se trouvent dans la partie de droite. D'après les formules de l'analyse combinatoire, on a

$$W = \binom{N}{k} = \frac{N!}{k! \cdot (N-k)!}$$

possibilités différentes. Le défaut d'information est en l'occurrence donné par

$I = ld(W) = ld\binom{N}{k}$ et l'entropie vaut de fait $S = \log(w) = k \cdot log\binom{N}{k}$

Le processus temporel est représenté graphiquement dans un graphique *GPanel*. On peut clairement observer que les particules se répartissent avec le temps entre les deux compartiments bien qu'il existe une infime chance que toutes les particules se retrouvent à un moment donné toutes à gauche. La probabilité de cet événement décroît cependant rapidement avec le nombre de particules présentes dans le système. La deuxième loi de la thermodynamique repose de ce fait sur une propriété statistique des systèmes comportant de nombreuses particules.

Étant donné que l'on n'observe jamais le processus inverse se produire, les processus de ce type sont dits **irréversibles**. Il existe cependant des systèmes passant du désordre à l'ordre mais il faut pour cela une « contrainte ordonnatrice ».

Some examples are:

Voici quelques exemples parlants de systèmes qui passent du désordre vers l'ordre

- La matière du cosmos, les étoiles et les systèmes planétaires
- La vie (matière très organisée) qui émerge de la matière non vivante
- Une contrainte suffisante permet de diminuer le niveau de désordre d'une cuisine, d'une salle de classe ou d'une chambre à couche
- Les nuages (liquides) et la glace (solide) émergent lors du refroidissement de la vapeur d'eau (gaz). De ce fait, les changements d'états modifient l'état d'ordre
- Les personnes assistant à un concert regagnent leurs places après la pause.

Dans tous ces processus, le niveau de désordre ainsi que l'entropie diminuent de sorte que des structures visiblement ordonnées émergent du chaos.

EXERCICES

1. Modifier le programme de simulation de gaz de sorte qu'après un certain laps de temps (59 secondes), un « démon » fait en sorte que le trou de la paroi de séparation ne laisse passer les particules que de droite à gauche mais pas de gauche à droite. Montrer que cela va faire baisser l'entropie du système.

2. Trouver d'autres exemples de systèmes qui:

   a. passent d'un état ordonné à un état désordonné

   b. passent d'un état désordonné à un état ordonné. Mettre en évidence la contrainte ordonnatrice.

chapitre onze

ANNEXES

Objectifs d'apprentissage

- Approfondir ses connaissances des algorithmes et de leur implémentation en *Python*.
- Avoir conscience des contextes propices aux erreurs de programmation les plus fréquentes.
- Connaître les principales techniques de débogage.
- Savoir ce que représente le traitement parallèle et être capable d'implémenter des programmes simples impliquant plusieurs fils d'exécution (Threads).
- Être capable de décrire ce qu'est une situation de compétition (race condition) et d'interblocage (deadlock).

11.1 JEUX LOGIQUES AMUSANTS

INTRODUCTION

Les jeux logiques comportant un aspect mathématique sont très populaires et répandus. Le but de ce chapitre n'est pas de vous ôter le plaisir de résoudre ce genre de jeux à la main mais bien plutôt de mettre en évidence que l'ordinateur peut en trouver la solution de manière systématique à l'aide du **retour sur trace** (*backtracking* en anglais). Cela se fait néanmoins avec deux différences significatives. D'une part, en l'absence de stratégie particulière et de limite claire, l'exploration systématique par l'ordinateur peut prendre un temps colossal, même sur un ordinateur très puissant. D'autre part, l'ordinateur est capable de trouver toutes les solutions possibles, ce qui n'est souvent pas faisable à la main. Ceci permet d'exploiter l'ordinateur pour vérifier qu'une certaine solution est la meilleure et la plus courte.

SUDOKU

Le Sudoku a subi un gros boom depuis 1980 et n'a cessé de se répandre depuis pour devenir très populaire de nos jours et se retrouver dans pratiquement tous les quotidiens ou hebdomadaires. Il s'agit d'un jeu numérique dont les règles sont très simples. Dans la version standard, les nombres de 1 à 9 doivent être placés dans une grille 9x9 de sorte que chaque nombre n'apparaisse qu'une seule fois dans chaque ligne et chaque colonne. De plus, la grille est divisée en neuf sous-grilles de 3x3 cellules dans lesquels chaque nombre doit apparaître exactement une seule fois. La donnée du jeu consiste en une grille partiellement remplie. Idéalement, une grille initiale ne devrait comporter qu'une et une seule solution.

Selon la configuration initiale, le jeu peut être plus ou moins difficile à résoudre. Un joueur expérimenté use de certaines stratégies bien connues et d'autres plus personnelles pour trouver la solution. Dans l'algorithme du retour sur trace naïf effectué par l'ordinateur, aucune stratégie n'est mise en place: les cellules vides sont remplies une à une avec les nombres de 1 à 9 de telle sorte qu'il n'y ait pas de conflit selon les règles du jeu. Si l'algorithme se retrouve dans une impasse, le dernier essai est annulé.

Dans le programme suivant, l'ordinateur utilise le backtracking. La grille *GameGrid* est idéale pour la représentation graphique puisque le jeu a une structure de grille. On dessiner les nombres initiaux en noir comme des *TextActor* et on insère la solution en rouge.

5	6	3	7	9	8	4	1	2
7	9	4	1	2	5	3	6	8
2	8	1	4	6	3	9	5	7
3	4	7	2	1	9	5	8	6
9	5	6	3	8	7	2	4	1
8	1	2	5	4	6	7	3	9
6	3	9	8	7	4	1	2	5
1	7	5	6	3	2	8	9	4
4	2	8	9	5	1	6	7	3

Comme vous le savez à propos du backtracking depuis le **chapitre 10.3.**, il faut d'abord trouver une structure de données appropriée pour stocker les états du jeu. Comme les états forment une grille 9x9, il faut choisir une liste composée de neufs listes représentant chacune une ligne. Le nombre 0 représente une cellule vide. De ce fait, la grille représentée ci-contre est représentée par l'état initial suivant:

```
startState = [ \
[0, 6, 0, 7, 9, 8, 0, 1, 2],
[7, 9, 4, 1, 0, 5, 0, 6, 8],
[2, 0, 1, 4, 0, 0, 9, 5, 7],
[0, 0, 0, 2, 1, 0, 5, 0, 0],
```

```
[0, 5, 6, 3, 0, 0, 2, 4, 1],
[0, 1, 2, 5, 4, 0, 7, 3, 9],
[6, 3, 0, 8, 7, 4, 0, 0, 0],
[1, 0, 5, 6, 0, 2, 8, 0, 0],
[4, 2, 8, 9, 0, 1, 0, 7, 0]]
```

Comme d'habitude, développons le programme étape après étape. La première tâche consiste à afficher un état donné dans la *GameGrid* et permettre à l'utilisateur de placer un nombre dans une cellule vide à l'aide de la souris. Bien que cette fonctionnalité ne soit pas nécessaire dans une résolution automatique du jeu, elle permet d'influencer le jeu de manière interactive, ce qui peut se révéler très utile dans la phase de test. Cela permet également de résoudre la grille à l'écran au lieu d'utiliser le papier et le crayon.

```
from gamegrid import *

def pressEvent(e):
    loc = toLocationInGrid(e.getX(), e.getY())
    if loc in fixedLocations:
        setStatusText("Location fixed")
        return
    x = loc.x
    y = loc.y
    value = startState[y][x]
    value = (value + 1)  % 10
    startState[y][x] = value
    showState(startState)

def showState(state):
    removeAllActors()
    for y in range(9):
        for x in range(9):
            loc = Location(x, y)
            value = state[y][x]
            if  value != 0:
                if loc in fixedLocations:
                    c = Color.black
                else:
                    c = Color.red
                digit = TextActor(str(value), c, Color.white,
                        Font("Arial", Font.BOLD, 20))
                addActorNoRefresh(digit, loc)
    refresh()

makeGameGrid(9, 9, 50, Color.red, False, mousePressed = pressEvent)
show()
setTitle("Sudoku")
setBgColor(Color.white)
getBg().setLineWidth(3)
getBg().setPaintColor(Color.red)
for x in range(4):
    getBg().drawLine(150 * x, 0, 150 * x, 450)
for y in range(4):
    getBg().drawLine(0, 150 * y, 450, 150 * y)

startState = [
[0, 6, 0, 7, 9, 8, 0, 1, 2],
[7, 9, 4, 1, 0, 5, 0, 6, 8],
[2, 0, 1, 4, 0, 0, 9, 5, 7],
[0, 0, 0, 2, 1, 0, 5, 0, 0],
[0, 5, 6, 3, 0, 0, 2, 4, 1],
[0, 1, 2, 5, 4, 0, 7, 3, 9],
[6, 3, 0, 8, 7, 4, 0, 0, 0],
[1, 0, 5, 6, 0, 2, 8, 0, 0],
[4, 2, 8, 9, 0, 1, 0, 7, 0]]
fixedLocations = []
for x in range(9):
    for y in range(9):
```

```
        if startState[y][x] != 0:
            fixedLocations.append(Location(x, y))
showState(startState)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

La prochaine étape consiste à définir une fonction *isValid(state)* qui vérifie qu'un état donné soit conforme aux règles du jeu. Il s'agit d'un travail fastidieux car il faut vérifier les 9 lignes, les 9 colonnes ainsi que les 9 sous-grilles 3x3. L'état de validité est affiché dans la barre d'état.


```
from gamegrid import *

def pressEvent(e):
    loc = toLocationInGrid(e.getX(), e.getY())
    if loc in fixedLocations:
        setStatusText("Location fixed")
        return
    xs = loc.x // 3
    ys = loc.y // 3
    x = loc.x % 3
    y = loc.y % 3
    value = startState[ys][xs][y][x]
    value = (value + 1)  % 10
    startState[ys][xs][y][x] = value
    showState(startState)
    if isValid(startState):
        setStatusText("State valid")
    else:
        setStatusText("Invalid state")

def showState(state):
    removeAllActors()
    for ys in range(3):
        for xs in range(3):
            for y in range(3):
                for x in range(3):
                    loc = Location(x + 3 * xs, y + 3 * ys)
                    value = state[ys][xs][y][x]
                    if  value != 0:
                        if loc in fixedLocations:
                            c = Color.black
                        else:
                            c = Color.red
                        digit = TextActor(str(value), c, Color.white,
                                Font("Arial", Font.BOLD, 20))
                        addActorNoRefresh(digit, loc)
    refresh()

def isValid(state):
    # Check lines
    for ys in range(3):
        for y in range(3):
            line = []
```

```
            for xs in range(3):
                for x in range(3):
                    value = state[ys][xs][y][x]
                    if value > 0 and value in line:
                        return False
                    else:
                        line.append(value)
    # Check rows
    for xs in range(3):
        for x in range(3):
            row = []
            for ys in range(3):
                for y in range(3):
                    value = state[ys][xs][y][x]
                    if value > 0 and value in row:
                        return False
                    else:
                        row.append(value)

    # Check subgrids
    for ys in range(3):
        for xs in range(3):
            subgrid = state[ys][xs]
            square = []
            for y in range(3):
                for x in range(3):
                    value = subgrid[y][x]
                    if value > 0 and value in square:
                        return False
                    else:
                        square.append(value)
    return True

makeGameGrid(9, 9, 50, Color.red, False, mousePressed = pressEvent)
show()
setTitle("Sudoku")
addStatusBar(30)
visited = []
setBgColor(Color.white)
getBg().setLineWidth(3)
getBg().setPaintColor(Color.red)
for x in range(4):
    getBg().drawLine(150 * x, 0, 150 * x, 450)
for y in range(4):
    getBg().drawLine(0, 150 * y, 450, 150 * y)

stateWiki = [[[[0, 3, 0], [0, 0, 0], [0, 0, 8]],
              [[0, 0, 0], [1, 9, 5], [0, 0, 0]],
              [[0, 0, 0], [0, 0, 0], [0, 6, 0]]],
             [[[8, 0, 0], [4, 0, 0], [0, 0, 0]],
              [[0, 6, 0], [8, 0, 0], [0, 2, 0]],
              [[0, 0, 0], [0, 0, 1], [0, 0, 0]]],
             [[[0, 6, 0], [0, 0, 0], [0, 0, 0]],
              [[0, 0, 0], [4, 1, 9], [0, 0, 0]],
              [[2, 8, 0], [0, 0, 5], [0, 7, 0]]]]
startState = stateWiki

fixedLocations = []
for xs in range(3):
    for ys in range(3):
        for x in range(3):
            for y in range(3):
                if startState[ys][xs][y][x] != 0:
                    fixedLocations.append(Location(x + 3 * xs, y + 3 * ys))

showState(startState)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

Pour effectuer le retour sur trace (backtracking), il est nécessaire de déterminer les sommets adjacents dans le graphe de transition des états du jeu avec la fonction *getNeighbours(state)*. Pour ce faire, on choisit une cellule vide avec *getEmptyCell()* et on insère dans l'ordre tous les nombres appartenant à un état du jeu autorisé. Au moins l'un de ces nombres sera à coup sûr le bon.

Dans la fonction *getNeighbours(state),* on commence par cloner la grille partielle reçue en paramètre dans une deuxième liste *clone* à l'aide de *cloneState(state)*. Cela est nécessaire pour pouvoir travailler sans danger dans cette grille puisqu'elle est passée par référence. Ne pas la cloner reviendrait à modifier la liste originale dans la partie appelante, ce qu'il faut à tout prix éviter dans ce cas (Voir la section « Modèle mémoire en Python » de ce même chapitre pour de plus amples explications à ce sujet). Ensuite, on choisit avec *getEmptyCell(state)* la prochaine cellule vide dans l'ordre de parcours de la grille choisi (en l'occurrence ligne après ligne, de haut en bas et de gauche à droite, selon la boucle imbriquée dans *getEmptyCell*). Chacun des neufs nouveaux états ainsi générés est ensuite cloné dans la liste d'états possibles *validStates* si *isValid()* indique qu'il est valide.

La fonction récursive *search()* qui effectue la recherche par backtracking peut être reprise pratiquement telle quelle des précédents problèmes résolus par backtracking puisque la partie dépendante du problème réside dans la fonction *getNeighbours(state).* Dans le cas présent, on ne cherche qu'une seule solution (les grilles de Sudoku bien formées ne possèdent qu'une solution) et on termine la récursion avec le fanion found puisqu'il est vain de poursuivre la recherche.


```
from gamegrid import *

def pressEvent(e):
    loc = toLocationInGrid(e.getX(), e.getY())
    if loc in fixedLocations:
        setStatusText("Location fixed")
        return
    x = loc.x
    y = loc.y
    value = startState[y][x]
    value = (value + 1)  % 10
    startState[y][x] = value
    showState(startState)
    if isValid(startState):
        setStatusText("State valid")
    else:
        setStatusText("Invalid state")

def getBlockValues(state, x, y):
    return [state[y][x], state[y][x + 1], state[y][x + 2],
            state[y + 1][x], state[y + 1][x + 1], state[y + 1][x + 2],
            state[y + 2][x], state[y + 2][x + 1], state[ y + 2][x + 2]]
```

```
def showState(state):
    removeAllActors()
    for y in range(9):
        for x in range(9):
            loc = Location(x, y)
            value = state[y][x]
            if  value != 0:
                if loc in fixedLocations:
                    c = Color.black
                else:
                    c = Color.red
                digit = TextActor(str(value), c, Color.white,
                        Font("Arial", Font.BOLD, 20))
                addActorNoRefresh(digit, loc)
    refresh()

def isValid(state):
    # Check lines
    for y in range(9):
        values = []
        for x in range(9):
            value = state[y][x]
            if value > 0 and value in values:
                return False
            else:
                values.append(value)
    # Check rows
    for x in range(9):
        values = []
        for y in range(9):
            value = state[y][x]
            if value > 0 and value in values:
                return False
            else:
                values.append(value)

    # Check blocks
    for yblock in range(3):
        for xblock in range(3):
            values = []
            li = getBlockValues(state, 3 * xblock, 3 * yblock)
            for value in li:
                if value > 0 and value in values:
                    return False
                else:
                    values.append(value)
    return True

def getEmptyCell(state):
    emptyCells = []
    for y in range(9):
        for x in range(9):
            if state[y][x] == 0:
                return [x, y]
    return []

def cloneState(state):
    li = []
    for y in range(9):
        line = []
        for x in range(9):
           line.append(state[y][x])
        li.append(line)
    return li

def getNeighbours(state):
    clone = cloneState(state)
    cell = getEmptyCell(state)
    validStates = []
```

```
    for value in range(1, 10):
        clone[cell[1]][cell[0]] = value
        if isValid(clone):
            validStates.append(cloneState(clone))
    return validStates

def search(state):
    global found, solution
    if found:
        return
    visited.append(state)  # state marked as visited

    # Check for solution
    if getEmptyCell(state) == []:
        solution = state
        found = True
        return

    for neighbour in getNeighbours(state):
        if neighbour not in visited: # Check if already visited
            search(neighbour) # recursive call
    visited.pop()

makeGameGrid(9, 9, 50, Color.red, False, mousePressed = pressEvent)
show()
setTitle("Sudoku")
addStatusBar(30)
visited = []
setBgColor(Color.white)
getBg().setLineWidth(3)
getBg().setPaintColor(Color.red)
for x in range(4):
    getBg().drawLine(150 * x, 0, 150 * x, 450)
for y in range(4):
    getBg().drawLine(0, 150 * y, 450, 150 * y)

startState = [
[0, 6, 0, 7, 9, 8, 0, 1, 2],
[7, 9, 4, 1, 0, 5, 0, 6, 8],
[2, 0, 1, 4, 0, 0, 9, 5, 7],
[0, 0, 0, 2, 1, 0, 5, 0, 0],
[0, 5, 6, 3, 0, 0, 2, 4, 1],
[0, 1, 2, 5, 4, 0, 7, 3, 9],
[6, 3, 0, 8, 7, 4, 0, 0, 0],
[1, 0, 5, 6, 0, 2, 8, 0, 0],
[4, 2, 8, 9, 0, 1, 0, 7, 0]]

fixedLocations = []
for x in range(9):
    for y in range(9):
        if startState[y][x] != 0:
            fixedLocations.append(Location(x, y))

showState(startState)
setStatusText("Press any key to search solution.")
getKeyCodeWait(True)
setStatusText("Searching. Please wait...")
found = False
solution = None
search(startState)
if solution != None:
    showState(solution)
    setStatusText("Solution found")
else:
    setStatusText("No solution")
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Il est possible d'utiliser ce programme pour résoudre n'importe quel Sudoku publié sur le Web ou dans un journal. Avec suffisamment de patience, le programme finira toujours par trouver la solution.

On pourrait diminuer le temps de résolution en incluant les stratégies de résolution utilisées lors d'une résolution à la main. À vous d'améliorer l'algorithme de recherche en y incluant de telles heuristiques de recherche.

La méthode de résolution utilisée peut très bien servir également à résoudre des Sudokus dont les cellules ne sont pas carrées. Il suffit pour cela d'adapter la fonction *getBlockValues()* en conséquence.

LES VILAINS MARIS JALOUX

En 1613 déjà, le mathématicien C. G. Bachet, Sieur de Méziriac, publia une étude intitulée *Problèmes plaisants et détectables qui se font par les nombres*, dans laquelle il décrit le jeu logique nommé *Les vilains maris jaloux* dont voici la description :

"Trois maris jaloux se trouvent le soir avec leurs femmes au passage d'une rivière, et rencontrent un bateau sans batelier; le bateau est si petit, qu'il ne peut porter plus de deux personnes à la fois. On demande comment ces six personnes passeront de tel sorte qu'aucune femme ne demeure en la compagnie d'un ou de deux hommes, si son mari n'est présent, soit sur l'une des deux rives, soit sur le bateau." (Lit. Édouard Lucas, L'arithmétique amusante" 1885, reprint 2006)

Claude Gaspard Bachet de Méziriac (1581-1638) (© Wiki)

Pour résoudre ce problème, on procède à nouveau étape par étape. On commence par créer l'interface graphique (une grille *GameGrid* fera l'affaire) puisque les personnes impliquées peuvent facilement être représentées par des images de sprite. On ajoute les acteurs à une grille invisible 7x3, ce qui permet de dessiner une rivière sur la bande du milieu. Puisque la seule chose qui importe est de savoir si une personne donnée est à gauche ou à droite de la rivière, on choisit comme structure de données un nombre binaire dont chacun des bits représente l'état d'une personne bien précise. Un bit à 0 signifie que la personne est à gauche et un bit à 1 qu'elle est à droite de la rivière. On utilise le bit de poids fort (celui ayant la plus grande valeur) pour représenter la position du bateau.

Dans le programme de simulation, il y a un couple de chaque couleur (rouge, vert, bleu) ainsi que le bateau qui sont associés de la manière suivante aux bits d'état:

b6	b5	b4	b3	b2	b1	b0
bateau	H_rouge	F_rouge	H_vert	F_verte	H_bleu	F_bleue

où b0 est le bit de poids faible (ayant la plus faible valeur numérique). En interprétant l'état du jeu dans le système décimal, l'ensemble des nombres entre 0 et 127 représentent les 128 états possibles au niveau des positions.

Il est fortement recommandé d'interrompre la lecture pour se familiariser avec la représentation des états sous forme de nombres binaires à l'aide du programme de simulation. Il faut cliquer sur un personnage ou sur le bateau pour le faire passer de l'autre côté de la rivière. L'état binaire ainsi que sa valeur décimale sont alors affichés dans la barre d'état.

You already build a test here with *isStateAllowed(state)* to check if the current situation is legal according to the rules. (You could also first create a prototype without this test.)

```
from gamegrid import *

def pressEvent(e):
    global state
    loc = toLocationInGrid(e.getX(), e.getY())
    if loc in left_locations:
        actor = getOneActorAt(loc)
        if actor != None:
            x = 6 - actor.getX()
            y = actor.getY()
            actor.setLocation(Location(x, y))
    if loc in right_locations:
        actor = getOneActorAt(loc)
        if actor != None:
            x = 6 - actor.getX()
            y = actor.getY()
            actor.setLocation(Location(x, y))
    state = 0
    for i in range(7):
        loc = right_locations[i]
        actor = getOneActorAt(loc)
        if actor != None:
            state += 2**(6 - i)
    showState(state)

def stateToString(state):
    return str(bin(state)[2:]).zfill(7)

def showState(state):
    sbin = stateToString(state)
    for i in range(7):
        if sbin[i] == "0":
           actors[i].setLocation(left_locations[i])
        else:
           actors[i].setLocation(right_locations[i])
    setTitle("State: " + str(state) + ", bin: " + stateToString(state))
    if isStateAllowed(state):
        setStatusText("Situation allowed")
    else:
        setStatusText("Situation not allowed")
    refresh()

def isStateAllowed(state):
    print state
    stateStr = stateToString(state)
    mred = stateStr[1] == "1"
    fred = stateStr[2] == "1"
    mgreen = stateStr[3] == "1"
    fgreen = stateStr[4] == "1"
    mblue = stateStr[5] == "1"
    fblue = stateStr[6] == "1"

    if mred and not fred or not mred and fred: # mred/fred not together
        if not fred and (not mgreen or not mblue) or fred and (mgreen or mblue):
            return False
    if mgreen and not fgreen or not mgreen and fgreen:#mgreen/fgreen not together
        if not fgreen and (not mred or not mblue) or fgreen and (mred or mblue):
            return False
    if mblue and not fblue or not mblue and fblue:  # mblue/fblue not together
        if not fblue and (not mred or not mgreen) or fblue and (mred or mgreen):
            return False
    return True

makeGameGrid(7, 3, 50, None, False, mousePressed = pressEvent)
```

```
setBgColor(Color.white)
addStatusBar(30)
show()
actors = [Actor("sprites/boat.png"),
   Actor("sprites/man_0.png"), Actor("sprites/woman_0.png"),
   Actor("sprites/man_1.png"), Actor("sprites/woman_1.png"),
   Actor("sprites/man_2.png"), Actor("sprites/woman_2.png")]

left_locations =  [Location(2, 0),
                   Location(2, 1), Location(2, 2),
                   Location(1, 1), Location(1, 2),
                   Location(0, 1), Location(0, 2)]
right_locations = [Location(4, 0),
                   Location(4, 1), Location(4, 2),
                   Location(5, 1), Location(5, 2),
                   Location(6, 1), Location(6, 2)]

for i in range(7):
    addActorNoRefresh(actors[i], left_locations[i])
for i in range(3):
   getBg().fillCell(Location(3, i), Color.blue)
refresh()

startState = 0
showState(startState)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

Dans une deuxième étape de développement, on implémente l'algorithme de recherche par backtracking bien connu du **chapter 10.3.** Encore une fois, il faut déterminer tous les états possibles atteignables à partir de l'état courant *state* au sein d'une fonction *getNeighbours(state)* qui fonctionne de la manière suivante

1. On commence par déterminer si le bateau se situe sur la rive gauche (*state < 64*) ou sur la rive droite (*state >= 64)*.
2. On dresser dans *li_one* la liste de toutes les personnes qui sont en mesure de traverser la rivière en solo et dans *li_two* la liste des personnes pouvant traverser la rivière par équipes de deux.
3. Il faut encore s'assurer à l'aide de *removeForbiddenTransfers() qu'aucun transfert déterminé aux étapes précédentes n'implique une épouse et un homme d'un autre couple.*

La recherche par backtracking est implémentée de manière familière dans la fonction *search().* On copie ensuite les solutions possibles dans une liste *solutions* qu'il sera possible d'examiner une fois le processus de recherche terminé. Ceci permet de compter le nombre de solutions possibles et de déterminer celle qui implique le moins de traversées et qui peut être visualisée pas à pas avec n'importe quelle touche du clavier.

```
from gamegrid import *
import itertools

def pressEvent(e):
    global state
    loc = toLocationInGrid(e.getX(), e.getY())
    if loc in left_locations:
        actor = getOneActorAt(loc)
        if actor != None:
            x = 6 - actor.getX()
            y = actor.getY()
            actor.setLocation(Location(x, y))
    if loc in right_locations:
        actor = getOneActorAt(loc)
        if actor != None:
            x = 6 - actor.getX()
            y = actor.getY()
            actor.setLocation(Location(x, y))
    state = 0
```

```
    for i in range(7):
        loc = right_locations[i]
        actor = getOneActorAt(loc)
        if actor != None:
            state += 2**(6 - i)
    setTitle("State: " + str(state) + ", bin: " + stateToString(state))
    if isStateAllowed(state):
        setStatusText("Situation allowed")
    else:
        setStatusText("Situation not allowed")
    refresh()

def stateToString(state):
    return str(bin(state)[2:]).zfill(7)

def showState(state):
    sbin = stateToString(state)
    for i in range(7):
        if sbin[i] == "0":
           actors[i].setLocation(left_locations[i])
        else:
           actors[i].setLocation(right_locations[i])
    refresh()

def getTransferInfo(state1, state2):
    state1 = state1 & 63
    state2 = state2 & 63
    mod = state1 ^ state2
    passList = []
    for n in range(6):
        if mod % 2 == 1:
            if n // 2 == 0:
                couple = "blue"
            elif n // 2 == 1:
                couple = "green"
            elif n // 2 == 2:
                couple = "red"
            if n % 2 == 0:
               passList.append("f" + couple)
            else:
               passList.append("m" + couple)
        mod = mod // 2
    return passList

def getTransferSequence(solution):
    transferSequence = []
    oldState = solution[0]
    for state in solution[1:]:
        transferSequence.append(getTransferInfo(oldState, state))
        oldState = state
    return transferSequence

def isStateAllowed(state):
    stateStr = stateToString(state)
    mred = stateStr[1] == "1"
    fred = stateStr[2] == "1"
    mgreen = stateStr[3] == "1"
    fgreen = stateStr[4] == "1"
    mblue = stateStr[5] == "1"
    fblue = stateStr[6] == "1"

    if mred and not fred or not mred and fred:  # mred/fred not together
        if not fred and (not mgreen or not mblue) or fred and (mgreen or mblue):
            return False
    if mgreen and not fgreen or not mgreen and fgreen:#mgreen/fgreen not together
        if not fgreen and (not mred or not mblue) or fgreen and (mred or mblue):
            return False
    if mblue and not fblue or not mblue and fblue:  # mblue/fblue not together
        if not fblue and (not mred or not mgreen) or fblue and (mred or mgreen):
```

```
            return False
    return True

def removeForbiddenTransfers(li):
    forbiddenPairs = [(0, 3), (0, 5), (1, 2), (2, 5), (1, 4), (3, 4)]
    allowedPairs = []
    for pair in li:
        if pair not in forbiddenPairs:
            allowedPairs.append(pair)
    return allowedPairs

def getNeighbours(state):
    neighbours = []
    li_one = []  # one person in boat
    bin = stateToString(state)
    if state < 64:  # boat at left
        for i in range(6):
            if bin[6 - i] == "0":
                li_one.append(i)
        li_two = list(itertools.combinations(li_one, 2)) #two persons in boat
        li_two = removeForbiddenTransfers(li_two)
    else: # boat at right
        for i in range(6):
            if bin[6 - i] == "1":
                li_one.append(i)
        li_two = list(itertools.combinations(li_one, 2))
        li_two = removeForbiddenTransfers(li_two)

    li_values = []
    if state < 64:  # boat at left, restrict to two persons transfer
        for li in li_two:
            li_values.append(2**li[0] + 2**li[1] + 64)
    else:  # boat at right, one or two persons transfer
        for i in li_one:
            li_values.append(2**i + 64)
        for li in li_two:
            li_values.append(2**li[0] + 2**li[1] + 64)

    for value in li_values:
        v = state ^ value
        if isStateAllowed(v):  # restrict to allowed states
            neighbours.append(v)
    return neighbours

def search(state):
    visited.append(state)  # state marked as visited

    # Check for solution
    if state == targetState:
        solutions.append(visited[:])

    for neighbour in getNeighbours(state):
        if neighbour not in visited: # Check if already visited
            search(neighbour) # recursive call
    visited.pop()

nbSolution = 0
makeGameGrid(7, 3, 50, None, False, mousePressed = pressEvent)
addStatusBar(30)
setBgColor(Color.white)
setTitle("Searching...")
show()
visited = []
actors = [Actor("sprites/boat.png"),
   Actor("sprites/man_0.png"), Actor("sprites/woman_0.png"),
   Actor("sprites/man_1.png"), Actor("sprites/woman_1.png"),
   Actor("sprites/man_2.png"), Actor("sprites/woman_2.png")]

left_locations =  [Location(2, 0),
```

```
                    Location(2, 1), Location(2, 2),
                    Location(1, 1), Location(1, 2),
                    Location(0, 1), Location(0, 2)]
right_locations = [Location(4, 0),
                    Location(4, 1), Location(4, 2),
                    Location(5, 1), Location(5, 2),
                    Location(6, 1), Location(6, 2)]

for i in range(7):
    addActorNoRefresh(actors[i], left_locations[i])
for i in range(3):
   getBg().fillCell(Location(3, i), Color.blue)
refresh()

startState = 0
targetState = 127
solutions = []
search(startState)

maxLength = 0
maxSolution = None
minLength = 100
minSolution = None
for solution in solutions:
    if len(solution) > maxLength:
        maxLength = len(solution)
        maxSolution = solution
    if len(solution) < minLength:
        minLength = len(solution)
        minSolution = solution
setStatusText("#Solutions: " + str(len(solutions)) + ", Min Length: "
              + str(minLength) + ", Max Length: " + str(maxLength))

setTitle("Press key to cycle")
oldState = startState
for state in minSolution[1:]:
    getKeyCodeWait(True)
    showState(state)
    info = getTransferInfo(oldState, state)
    setTitle("#Transferred: " + str(info))
    oldState = state
setTitle("Done. #Boat Transfers: " + str((len(minSolution) - 1)))
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

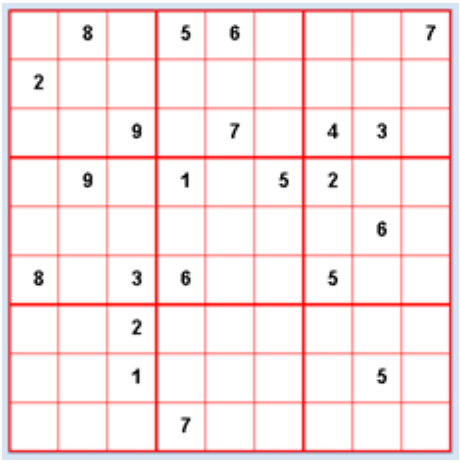
MEMENTO

Avec de tels casse-têtes, on ne sait jamais à l'avance s'ils possèdent une ou plusieurs solutions. Il se trouve qu'il est bien plus facile, pour nous humains, de trouver une solution en tâtonnant que de prouver qu'il n'existe pas de solution. Par contre, pour un ordinateur qui effectue une recherche exhaustive de manière systématique, l'énumération de toutes les configurations possibles et de ce fait la recherche de l'ensemble des solutions possibles n'est pas un problème, même si celle-ci peut nécessiter un temps énorme. Cette situation peut malheureusement déjà survenir avec des casse-têtes relativement simples en raison de l'explosion combinatoire, de sorte que l'on touche à nouveau aux limites de la calculabilité et de l'utilisation de l'ordinateur.

Il est intéressant de noter que Bachet avait déjà pu trouver la solution optimale impliquant 11 traversées, ce qui correspond à ce que trouve notre programme. Il développe une argumentation tellement géniale traversée après traversée que la stratégie de résolution du casse-tête en devient évidente. On ne sait par contre pas s'il a d'abord utilisé une approche par tâtonnement et établi seulement par la suite son argumentation ou s'il a pu directement élaborer la solution optimale.

EXERCICES

1. Résoudre le X-Sudoku suivant. Le X-Sudoku implique comme règle supplémentaire que les nombres 1 à 9 ne doivent apparaître qu'une seule fois dans chacune des grandes diagonales de la grille.



	8		5	6				7
2								
		9		7		4	3	
	9		1		5	2		
							6	
8		3	6			5		
		2						
		1					5	
			7					

2. Montrer qu'il n'est pas possible de résoudre le problème des « vilains maris jaloux » si uniquement une personne à la fois peut effectuer la traversée.

11.2 TRAPPES, RÈGLES ET ASTUCES

INTRODUCTION

Comme c'est le cas dans de nombreux langages de programmation, le langage Python recèle quelques points délicats auxquels même les programmeurs expérimentés ont à faire face. Vous pouvez vous aussi être en mesure de gérer ces difficultés si vous avez conscience qu'ils peuvent constituer une source de danger potentiel.

LE MODÈLE MÉMOIRE EN PYTHON

Dans le chapitre 2.6, nous avons abordé les variables en se les représentant comme des boîtes dans lesquelles on peut stocker des valeurs comme des nombres. En gros, cette représentation laisse pour le moment entendre que lors d'une définition de variable telle que *a = 2*, Python va réserver un espace dans la mémoire vive semblable à une boîte dans lequel le nombre 2 sera stocké. La variable *a* joue alors le rôle de symbole que l'on utilise à la place de la valeur 2.

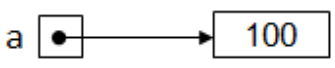
Cette représentation est bien pratique pour aborder les variables mais il faut savoir qu'elle est limitée et peut conduire à une compréhension erronée du programme car elle n'est pas vraiment correcte et ne rend pas compte de ce qui se passe réellement. Cela vient de ce que toutes les données, en particulier les nombres, sont considérées par Python comme des objets. Les objets sont des structures complexes qui sont bien plus que juste une valeur. Les objets présentent par exemple également des comportements (fonctions, méthodes). Les nombres entiers (objets de type *int*) « savent » ainsi comment s'additionner entre eux. Voici quelques lignes qui montrent le bien-fondé de cette affirmation:

```
>>> a = 100
 < a: 100
>>> a.__add__
 < built-in method __add__ of int object at 0x2>
```

Ainsi, chaque objet possède une identité (identifiant) unique qui peut être interrogé par la fonction intégrée *id()*:

```
>>> id(a)
 < 2
```

Il faut donc bien comprendre qu'en Python, lorsque l'on définit une variable *a*, ce nom *a* n'est rien d'autre qu'un nom qui fait référence à un objet de type *int* se trouvant en mémoire à l'emplacement mémoire (adresse) 0x2. On dit parfois que *a* est un alias (dans d'autres langages de programmation, on parle de *référence* ou de *pointeur*). Voici comment se représenter la situation de manière visuelle:



La différence avec la métaphore de la boîte utilisée jusqu'à présent apparaît avec l'affectation suivante :

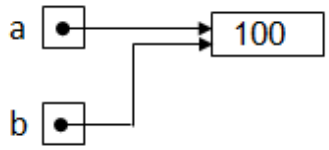
```
>>> b = a
 < b: 100
```

La métaphore de la boîte pourrait laisser penser qu'après cette affectation, la mémoire contient deux boîtes différentes contenant toutes deux la valeur 100. En réalité, il n'en est rien : *b* est

juste un deuxième alias faisant référence **au même objet mémoire** que *a*, ce qui confirme la fonction *id()* :

```
>>> is(b)
 < 2
```

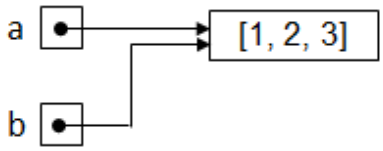
Visuellement, la situation est donc la suivante en mémoire:



Ce modèle mémoire orienté objet est d'une très grande importance lorsque le programme ne se contente pas d'utiliser des nombres mais qu'il fait un usage abondant de types de données structurées complexes telles que les listes. Pour s'en convaincre, commençons par définir une première liste *a* ainsi qu'un deuxième liste *b* faisant référence au même objet suite à une affectation:

```
>>> a = [1, 2, 3]
 < a: [1, 2, 3]
>>> b = a
 < b: [1, 2, 3]
```

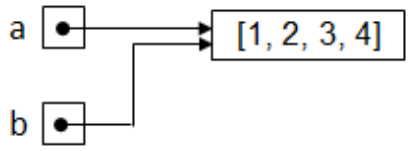
Comme le laisse entendre notre nouveau modèle mémoire, la situation est la suivante:



Ainsi, si l'on modifie la liste au travers de l'alias *b*

```
>>> b.append(4)
>>> b
 < [1, 2, 3, 4]
```

la situation en mémoire sera la suivante:



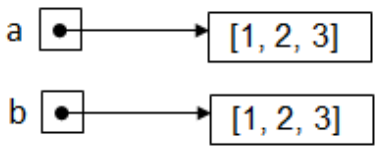
Il n'est donc pas étonnant que la liste référencée par l'alias *a* ait également changé!

```
>>> a
 < a: [1, 2, 3, 4]
```

Cette interdépendance des variables *a* et *b* peut conduire à d'innombrables erreurs de programmation très subtiles et souvent très difficiles à identifier. Cette interdépendance n'intervient cependant pas lorsque *b* est nouvellement définie comme une liste séparée car Python génère dans ce cas une nouvelle liste totalement indépendante:

```
>>> a = [1, 2, 3]
 < a: [1, 2, 3]
>>> b = a
>>> b = [1, 2, 3]
 < b: [1, 2, 3]
```

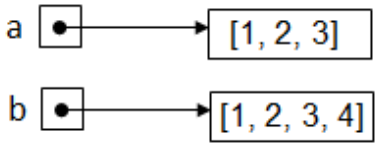
Visuellement, cela produit la situation suivante en mémoire:



ce qui permet de modifier la liste *b* sans affecter la liste *a*:

```
>>> b.append(4)
```

puisque l'on aura alors la situation suivante en mémoire:



On peut vérifier cette affirmation avec le code suivant:

```
>>> b
 < b: [1, 2, 3, 4]
>>> a
 < a: [1, 2, 3]
```

Cette interdépendance entre les deux variables *a* et *b* n'est pas perceptible lorsque l'on utilise des nombres entiers. En effet, lors de l'affectation d'une nouvelle valeur 200 à la variable *b*, un nouvel objet est créé en mémoire. L'alias *b,* qui pointait vers le même entier 100 que *a* après la deuxième instruction *b = a*, pointe sur le nouvel objet entier 200 après la troisième instruction.

```
>>> a = 100
 < a: 100
>>> b = a
>>> a
 < a: 100
```

Contrairement à l'idée que favorise la métaphore de la boîte utilisée jusqu'à présent pour se représenter le fonctionnement des variables, l'affectation d'une variable à une autre ne réalise pas de copie de la valeur.

Le code suivant montre encore une fois à quel point l'affectation à une nouvelle variable d'une liste contenant des chaines de caractères est problématique. En effet, l'affectation ne copie pas du tout le contenu de la liste:

```
>>> myGarden = ["Rose", "Lotus"]
>>> yourGarden = myGarden
>>> yourGarden[0] = "Hibiskus"
>>> myGarden
 < ["Hibiskus", "Lotus"]
>>> yourGarden
 < ["Hibiskus", "Lotus"]
```

On peut donc retenir la règle suivante:

Règle 1a:
Une copie à l'aide de l'opérateur d'affectation constitue généralement une erreur. Les types de données non mutables sont à ce titre une exception (voir ci-dessous).

Si l'on veut réaliser une véritable copie indépendante d'une liste, à savoir un clone, il y a essentiellement deux solutions à disposition. La première consiste à copier explicitement chaque

élément de la liste d'origine dans une autre liste à l'aide d'un code « maison ». La deuxième consiste à utiliser la fonction *deepcopy()* du module *copy* comme le montre l'exemple suivant

```
>>> import copy
>>> myGarden = ["Rose", "Lotus"]
>>> yourGarden = copy.deepcopy(myGarden)
>>> yourGarden[0] = "Hibiskus"
>>> myGarden
 < ["Rose", "Lotus"]
>>> yourGarden
 < ["Hibiskus", "Lotus"]
```

Règle 1b:
Pour éviter toute surprise lors de la copie d'un type de données mutable, il faut toujours en effectuer une véritable copie (clone) à l'aide de la fonction *copy.deepcopy()*.

On a vite fait d'enfreindre la règle 1 lorsque l'on passe des valeurs en paramètre à une fonction. Il se trouve en effet que si l'on passe en paramètre à une fonction un type de données non élémentaire et mutable tel qu'une liste, la fonction en question peut sans problème modifier le contenu de cette liste lors de son exécution:

```
>>> def show(garden)
>>>     print "garden:", garden
>>>     garden[0] = "Hibiskus"
>>> myGarden = ["Rose", "Lotus"]
>>> show(myGarden)
>>> myGarden
 < ["Hibiskus", "Lotus"]
```

Lors de l'appel de la fonction *show()*, le contenu de la liste *garden* a été modifié par la fonction durant son exécution. On appelle ce phénomène un **effet de bord** de l'appel de la fonction *show()* tout-à-fait similaire aux effets secondaires et indésirables d'un médicament dans le domaine pharmacologique.

Règle 2:
Modifier la valeur d'un paramètre de l'intérieur d'une fonction et produire des effets de bord ne constitue pas un signe de maturité en programmation. C'est un comportement dangereux assimilable à du « codage en état d'ébriété » qui peut engendrer un chaos incontrôlable dans le programme.

TYPES DE DONNÉES MUTABLES ET IMMUTABLES

Afin de réduire les risques liés aux effets de bord, les concepteurs du langage Python ont introduit une distinction entre types de données mutables et immutables. Les types de données immutables connus jusqu'à présent sont les suivants : les nombres entiers (int), les nombres flottants (float), les chaines de caractères (str), les bytes (byte) et les tuples. De ce fait, si l'on tente de modifier un caractère individuel à l'intérieur d'une chaine de caractères, le programme lève une exception

```
>>> s = "abject"
 < s: "abject"
>>> s[0] = "o"
 < TypeError:can't assign to immutable object
```

Pour corriger le contenu de la chaine de caractères *s,* il faut redéfinir la chaine à neuf :

```
>>> s = "abject"
 < s: "abject"
>>> s = "object"
 < s: "object"
```

Lors de ce procédé, un tout nouvel objet *str* est créé qui n'est aucunement lié à la précédente chaine. L'alias *s* pointe alors sur ce nouvel objet et l'ancien est « oublié » s'il n'est référencé par aucun autre alias. À l'interne, le **ramasse-miettes** de Python (*Garbage Collector* en anglais) supprime périodiquement les objets « défunts » ainsi oubliés lorsqu'ils ne sont plus référencés par aucun alias.

EMPAQUETAGE ET DÉPAQUETAGE

À première vue, les tuples ne semblent pas être très différents des listes. Ils sont en fait comme des listes immutables et supportent donc toutes les opérations présentes sur les listes qui n'en modifient pas le contenu. Il faut cependant connaître quelques détails de notation des tuples impliquant des virgules de manière spéciale.

On peut omettre les parenthèses rondes lors de la création d'un tuple:

```
>>>  t = 1, 2, 3
>>>  t
>>>  (1, 2, 3)
```

La virgule est dans ce cas utilisée comme un caractères syntaxique permettant de séparer les éléments les uns des autres. Cette technique appelée **empaquetage automatique** (*automatic packing* en anglais) peut être utilisée pour retourner plusieurs valeurs différentes sous forme de tuple depuis une fonction:

```
>>> import math
>>> def sqrt(x:
>>>     y = math.sqrt(x)
>>>     return y, -y
>>> sqrt(4)
 < (2,0, -2,0)
```

On peut également utiliser l'opérateur virgule dans la partie gauche d'une assignation dont l'expression droite retourne un tuple. On parle alors de **dépaquetage automatique** (***automatic unpacking* en anglais)**:

```
>>> import math
>>> def sqrt(x):
>>>     y = math.sqrt(x)
>>>     return y, -y
>>> y1, y2 = sqrt(2)
>>> y1
 < 1.41421356237330951
>>> y2
 < -1.41421356237330951
```

On peut par exemple utiliser cette technique pour définir rapidement plusieurs variables différentes de manière simultanée:

```
>>> a, b, c = 1, 2, 3
>>> a
 < 1
>>> b
 < 2
```

```
>>> c
 < 3
```

Dans l'exemple suivant, la première instruction effectue un empaquetage automatique dans l'expression de droite tandis que la deuxième réalise un dépaquetage automatique dans la partie gauche de l'assignation:

```
>>> t = 1, 2, 3
>>> a, b, c = t
>>> a
 < 1
>>> b
 < 2
>>> c
 < 3
```

Notons qu'il est également possible d'utiliser la technique du dépaquetage automatique avec les listes :

```
>>> li = [1, 2, 3]
>>> a, b, c = li
>>> a
 < 1
>>> b
 < 2
>>> c
 < 3
```

Cela permet par exemple de permuter de manière élégante les valeurs de deux variables sans aucune variable supplémentaire :

```
>>> li = [1, 2, 3]
>>> a, b = b, a
>>> a
 < 2
>>> b
 < 1
```

LISTES BIDIMENSIONNELLES, MATRICES

Les matrices sont réalisées par des tableaux dans de nombreux langages de programmation. Les lignes sont stockées sous forme de tableaux (listes) et la matrice en elle-même comme une liste de lignes, à savoir une liste de listes. Il est trivial d'utiliser des listes au lieu de tableaux en Python. Il faut cependant faire très attention au fait qu'en Python les listes ne se comportent pas comme des types de données élémentaires et immutables puisqu'elles ne font que de stocker des références sur d'autres objets et non les objets eux-mêmes. Lors de la création d'une matrice, on peut très facilement tomber dans un travers déjà rencontré et traité. Sans se douter de rien, voici comment un programmeur inattentif va générer une matrice 3x3 remplie de zéros dans la console Python.

```
>>> A = [[0] * 3] * 3
>>> A
 < [[0, 0, 0],
    [0, 0, 0],
    [0, 0, 0]]
```

Ce programmeur tête en l'air sera certainement surpris lorsqu'il tentera de modifier la dernière valeur de la première ligne à l'aide de l'opérateur d'affectation : il constatera en effet que la

dernière valeur a changé dans toutes les lignes.

```
>>> A[0][2] = 1
>>> A
 < [[0, 0, 1],
    [0, 0, 1],
    [0, 0, 1]]
```

Que s'est-il donc passé ? Il vous suffit sans doute d'y réfléchir attentivement en vous rappelant ce que l'on a déjà dit sur les copies de listes. Le code précédent est en effet parfaitement équivalent au code suivant qui commence par créer une variable *z*:

```
>>> z = [0] * 3
>>> A = [z] * 3
>>> A
 < [[0, 0, 0],
    [0, 0, 0],
    [0, 0, 0]]
```

On crée tout d'abord une liste *z* contenant trois zéros puis une liste *A* fabriquée à partir de cette même liste *z* répétée trois fois. Il faut cependant bien voir que ces trois listes ne sont que trois références à la même liste en mémoire. Ainsi, si l'on change l'une d'entre elles, les « autres » seront également affectées.

Règle 3:
Ne jamais utiliser la multiplication de listes pour générer des listes imbriquées.

Pour éviter cet écueil, on peut utiliser une compréhension de liste. Comme vous pourrez le vérifier, la matrice se comporte correctement si elle est créée de la manière suivante:

```
>>> A = [[0 for x in range(3)] for y in range(3)]
>>> A
 < [[0, 0, 0],
    [0, 0, 0],
    [0, 0, 0]]
>>> A[0][2] = 1
>>> A
 < [[0, 0, 1],
    [0, 0, 0],
    [0, 0, 0]]
```

FUNCTION DECORATORS

En Python, il est possible « décorer » les fonctions à l'aide d'une ligne débutant par le symbole @. Une telle ligne s'appelle un **décorateur** et on l'utilise pour associer des propriétés spéciales à une fonction. Un décorateur est donc une fonction qui fait office d'emballage (*function wrapper* en anglais) pour la fonction originale qui est décorée. La fonction décorée est en général appelée au sein de la fonction décorateur. Dans le programme suivant, la fonction *trisect(x)* est décorée de telle manière qu'elle retourne la valeur 0 pour x = 0 et que toutes les valeurs soient arrondies à deux décimales.

```
def tri(func):
    # inner function
    def _tri(x):
        if x == 0:
            return 0
        return round(func(x), 2)
    return _tri
```

```
@tri
def trisect(x):
    return 3 / x

for x in range(0, 11):
   value = trisect(x)
   print value
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

Nous avons déjà vu dans le chapitre **Chapitre 3.7** qu'il est possible de décorer des fonctions de telle manière qu'elles soient automatiquement enregistrées comme des gestionnaires d'événement sous forme de fonction de rappel. Dans le **Chapitre 7.2**, nous avons également utilisé le décorateur *@staticmethod* pour rendre les méthodes statiques.

PROGRAMMATION FONCTIONNELLE ET MODULAIRE

Nous avons déjà vu qu'il faut structurer un programme de manière à isoler les bouts de codes que l'on utilisera à de nombreuses reprises dans des fonctions. La plupart du temps, ces fonctions font appel à des valeurs supplémentaires que l'on peut leur fournir par des arguments ou par des variables globales. Le résultat du traitement effectué par la fonction est quant à lui retourné soit par l'intermédiaire d'une instruction *return* ou par une variable globale. Cependant, comme nous l'avons déjà vu dans la règle 2, le fait de modifier des variables globales depuis une fonction constitue un très mauvais style de programmation qui, en plus d'être très inélégant, est également très dangereux puisque la fonction produit alors des effets de bords, comparables aux effets secondaires indésirables des médicaments. Pour éviter ces dangers, on veillera à respecter la

Règle 4:
Pour éviter les problèmes, il faut à tout prix éviter d'utiliser des variables globales depuis les fonctions. Il est particulièrement dangereux de modifier des variables globales depuis une fonction et, de ce fait, de recourir au mot-clé *global* qui supprime la protection que Python permet contre le danger des effets de bord.

Le paradigme de **programmation fonctionnelle** consiste à n'utiliser que des fonctions qui ne produisent pas d'effet de bord. De telles fonctions comportent de surcroît l'avantage suivant : du fait qu'elles se suffisent à elles-mêmes et n'utilisent que les valeurs passées en paramètre, elles peuvent sans problème être placées dans un fichier séparé et, de ce fait, être réutilisées dans un autre programme. Un tel fichier, regroupant différentes fonctions et classes apparentées, est appelé **module**. Un projet informatique un tant soit peu sérieux est normalement séparé en plusieurs modules qui peuvent parfois être développés par plusieurs personnes ou équipes différentes. Un programme bien conçu utilise donc habituellement les principes de base de la **programmation modulaire**.

Comme les fonctions d'un module peuvent être réutilisées dans plusieurs domaines différents, on surnomme parfois les modules ou un ensemble de modules une **bibliothèque de programme**, ou simplement une **bibliothèque**. Du point de vue de l'utilisateur de la bibliothèque, les détails d'implémentation des fonctions ne sont pas importants. Le programmeur utilisateur doit uniquement être en mesure de pouvoir déterminer, à l'aide de la documentation de la bibliothèque, les noms des fonctions, la liste de paramètres qu'elles acceptent et la spécification des valeurs de retour possibles. Du fait que le fonctionnement exact des fonctions de la bibliothèque demeure caché au programmeur qui les utilise, on parle souvent de **boîtes noires**. [**plus...**].On pourrait également décrire ce paradigme de programmation par le fait qu'il existe un contrat entre le programmeur utilisateur U qui utilise la bibliothèque et le programmeur D qui la développe. Aussi longtemps que l'utilisateur U s'en tient aux préconditions supposées par les fonctions de la bibliothèque, celles-ci garantissent un comportement selon le contrat (postconditions) [**plus...**]

Dans la mesure où l'on développe les fonctions en respectant les principes de la programmation fonctionnelle (sans effet de bord), on peut placer ces fonctions dans un fichier séparé du

programme principal en les baptisant d'un nom significatif et évoquant clairement leur rôle. Pour les réutiliser depuis un autre programme, il est alors nécessaire d'importer ce module. En guise de démonstration, le programme suivant définit un module *starlib.py* permettant de dessiner des étoiles de différentes tailles à l'aide de la tortue. Au début du programme principal, les fonctions du module sont importées à l'aide de l'instruction *from starlib import* *. Si l'on ne fait que d'utiliser la fonction *star* du module *starlib*, on peut optimiser un peu l'importation en spécifiant explicitement la fonction à importer avec l'instruction *from starlib import star.* Cette forme d'importation spécifique est même meilleure puisqu'elle n'importera que les fonctions réellement nécessaires et utilisées.

```
# starlib.py

from gturtle import *

def star(size):
    startPath()
    for i in range(5):
        forward()
        left(144)
    fillPath()
```

Sélectionner le code

```
from gturtle import *
import random
from starlib import *

makeTurtle()
ht()
for i in range(500):
    setPos(random.randint(-400, 400),
           random.randint(-400, 400))
    star(random.randint(10, 50))
```

Sélectionner le code

Il existe deux manières différentes d'importer un module. On utilise soit *from ...* soit la version *import ...* Dans le deuxième cas, il faut cependant préfixer le nom de la fonction par le nom du module auquel elle appartient, séparés par un point. Le module importé doit se trouver dans le même dossier que le programme qui l'importe. Si vous désirez réutiliser un module destiné à être importé depuis des programmes se trouvant dans différents dossiers, le mieux est de copier ce module dans le dossier *Lib* situé au même endroit que l'archive *tigerjython2.jar* [**plus...**].

```
from gturtle import *
import random
import starlib

makeTurtle()
ht()
for i in range(500):
    setPos(random.randint(-400, 400),
       random.randint(-400, 400))
    starlib.star(random.randint(10, 50))
```

Sélectionner le code

11.3 BOGUES ET DEBOGUAGE

INTRODUCTION

La perfection absolue n'est pas de ce monde. On peut considérer que tout programme contient des imperfections. En informatique, celles-ci ne sont pas appelées *erreurs*, mais bogues (de l'anglais *bug* = insecte). Les bogues se manifestent lorsque le programme réagit d'une manière non conforme à ce qui est attendu ou s'il se plante. De ce fait, les compétences de déboguage sont aussi vitales que celles permettant d'écrire du code. Il est bien clair entre nous que les codeurs devraient faire de leur mieux pour éviter d'introduire des bogues dans leur programme. Sachant que chaque ligne de code est susceptible de créer un bogue, il faut user de prudence et programmer de manière défensive. Par exemple, lorsque l'on n'est pas totalement convaincu qu'un bout de code ou un algorithme est correct, il est préférable de tester ces lignes indépendamment du reste du code et de se concentrer sur cette partie du code au lieu de foncer la tête dans le guidon en laissant ce travail pour plus tard. De nos jours, quantité de composants logiciels sont responsables de gérer de grandes sommes d'argent ou des appareils mettant des vies en jeu. En tant que développeur de composant logiciel assumant un rôle aussi crucial, il est nécessaire d'assumer la responsabilité de tester chaque ligne de code pour garantir qu'elle fonctionne correctement. La bricole rapide et le tâtonnement sont totalement exclus dans ce genre de cas.

L'objectif crucial du développement d'algorithmes et de gros systèmes logiciels est de produire des programmes qui sont autant que faire se peut dépourvus de bogues. Il existe de nombreuses approches pour parvenir à ce but. On peut utiliser des raisonnements mathématiques pour tenter de démontrer que le programme possède le comportement désiré. Cela peut se faire sans ordinateur, en utilisant des raisonnements mathématiques similaires à ceux utilisés pour prouver la validité d'un théorème. Cela n'est cependant possible que pour des programmes relativement simples. Une autre possibilité consiste à limiter la syntaxe du langage de programmation pour empêcher le programmeur de faire des choses qui pourraient s'avérer dangereuses. Un des exemples typiques a consisté à supprimer des langages modernes la notion de pointeurs présents dans les langages bas niveau et rendant possible le fait de référencer les faux objets en mémoire ou des emplacements mémoires indéfinis. C'est la raison pour laquelle il existe des langages qui imposent volontairement des restrictions de ce type et d'autres, moins sécurisés, qui laissent une marge de manœuvre considérable au programmeur. Le langage Python appartient plutôt à la famille des langages permissifs et adopte la maxime suivante

"We are all adults and decide for ourselves what we do and what we shouldn't."
or *"After all, we are all consenting adults here".*
« Nous sommes tous des adultes et décidons par nous-mêmes ce que l'on peut faire et ce qu'il ne fait pas faire. » ou « Après tout, nous sommes tous des adultes consentants ».

Un principe important pour créer des programmes qui sont aussi solides que possibles s'appelle la programmation par contrat (*Design by Contract (DbC)*). Celle-ci remonte au père du langage Eiffel, Bertrand Meyer, à l'ETH Zürich. Meyer envisage un logiciel comme étant un contrat entre un programmeur A et l'utilisateur B, où B pourrait très bien être un programmeur qui utilise les modules et bibliothèques fournis par le programmeur A. Sous forme de contrat, le programmeur A détermine très précisément sous quelles conditions (préconditions / prémisses) son module va fournir un résultat correct et décrit très précisément ce qui est retourné (les postconditions). En d'autres termes, l'utilisateur B se conforme aux préconditions exigées par A de telle sorte qu'il obtient de A la garantie que le résultat qu'il va obtenir est conforme aux postconditions annoncées. B n'a pas besoin de connaître les détails d'implémentation du module et A peut changer cette implémentation n'importe quand aussi longtemps que les postconditions sont respectées.

ASSERTIONS

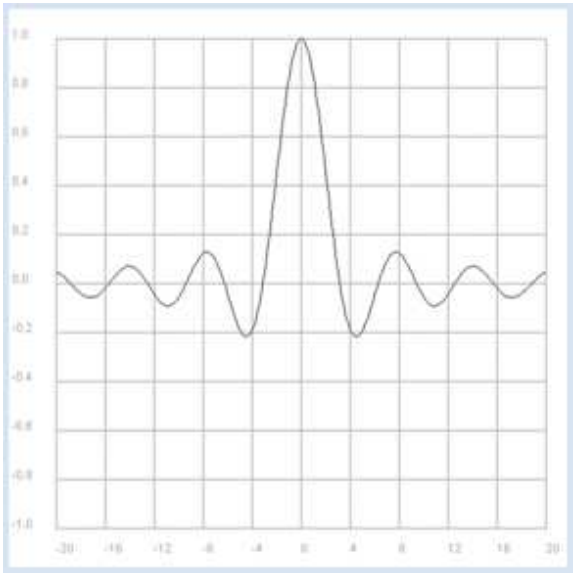
Du fait que le programmeur A, le producteur du module, ne fait pas trop confiance à l'utilisateur B, il va incorporer dans son code des tests qui permettent de vérifier que les préconditions nécessaires au bon fonctionnement sont bien remplies. On les appelle des **assertions**. Si ces assertions ne sont pas respectées, le programme va généralement quitter en levant une exception et en affichant un message d'erreur. On peut aussi imaginer que l'erreur soit gérée sans causer le crash du programme. Dans ce cas, il faut appliquer le principe suivant de génie logiciel :

Il est préférable de stopper l'exécution du programme avec un message d'erreur descriptif et mettant en évidence les causes possibles de l'erreur plutôt que de le laisser tourner pour aboutir à un résultat erroné.

Un programmeur A développe par exemple une fonction *sinc(x)* (sinus cardinalis) qui joue un rôle très important en traitement de signal. Voici la définition mathématique :

$$f(x) = \frac{\sin x}{x}$$

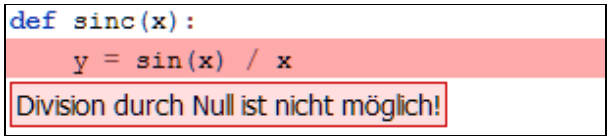
Un programmeur B veut l'utiliser pour réaliser une représentation graphique de la fonction pour *x* variant entre *-20* et *20*.



```
from gpanel import *
from math import pi, sin

def sinc(x):
    y = sin(x) / x
    return y

makeGPanel(-24, 24, -1.2, 1.2)
drawGrid(-20, 20, -1.0, 1, "darkgray")
title("Sinus Cardinalis: y = sin(x) / x")

x = -20
dx = 0.1
while x <= 20:
    y = sinc(x)
    if x == -20:
        move(x, y)
    else:
        draw(x, y)
    x += dx
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

Au premier coup d'œil, il n'y a pas de souci et le programme semble tourner correctement … dans ce cas précis … En effet, si l'on fixe l'incrément de *x* à 1, le programme rencontre une erreur malencontreuse :



Une des manières de résoudre ce problème consiste pour A à exiger la précondition que *x* ne soit pas nul et mettre en œuvre une assertion qui décrit précisément l'erreur.

```
from gpanel import *
from math import pi, sin

def sinc(x):
    if x == 0:
        return 1.0
    y = sin(x) / x
    return y

makeGPanel(-24, 24, -1.2, 1.2)
drawGrid(-20, 20, -1.0, 1, "darkgray")
title("Sinus Cardinalis: y = sin(x) / x")

x = -20
dx = 1
while x <= 20:
    y = sinc(x)
    if x == -20:
        move(x, y)
    else:
        draw(x, y)
    x += dx
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

Le message d'erreur est maintenant bien meilleur puisqu'il indique exactement où se situe l'erreur rencontrée.

Il serait encore bien mieux que le programmeur A traite spécifiquement le cas *x = 0* en retournant la valeur limite de la fonction lorsque x tend vers 0.

```
from gpanel import *
from math import pi, sin

def sinc(x):
    if x == 0:
        return 1.0
    y = sin(x) / x
    return y

makeGPanel(-24, 24, -1.2, 1.2)
drawGrid(-20, 20, -1.0, 1, "darkgray")
title("Sinus Cardinalis: y = sin(x) / x")

x = -20
dx = 1
while x <= 20:
    y = sinc(x)
    if x == -20:
        move(x, y)
    else:
        draw(x, y)
    x += dx
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

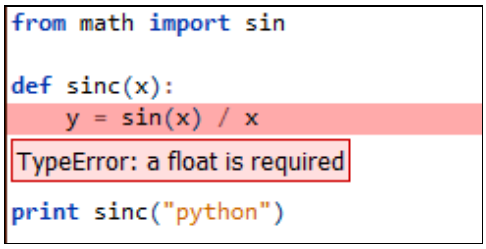
Ce code viole cependant la règle fondamentale qu'il faut toujours se méfier des nombres flottants dans les comparaisons à cause des inévitables erreurs d'arrondis effectuées par le processeur. Il serait encore mieux d'effectuer le test en utilisant une valeur de tolérance epsilon :

```
def sinc(x):
    epsilon = 1e-100
    if abs(x) < epsilon:
        return 1.0
```

Cette version de la fonction *sinc(x)* n'est toujours pas complètement bétonnée car il faudrait encore exiger que *x* soit bien un nombre réel. Si l'on s'aventurait par exemple à effectuer l'appel avec *x = "python"*, une erreur incongrue surviendrait immanquablement :

```
from math import sin

def sinc(x):
    y = sin(x) / x
    return y

print sinc("python")
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)



Ceci montre l'avantage des langages de programmation avec déclaration explicite du type de données des variables, ce qui permet une détection de ce genre de problème comme erreur de syntaxe avant son exécution (lors de la compilation / vérification syntaxique) et, de ce fait, avant qu'elle ne touche l'utilisateur.

AFFICHER DES INFORMATIONS DE DÉBOGUAGE

Les bons programmeurs sont reconnus pour leur capacité à éliminer les erreurs très rapidement [**plus...** Il existe un mouvement de développement logiciel basé sur la notion de développement dirigé par les tests (Test Driven Development (TDD) en anglais)]. Comme nous pouvons tous apprendre de nos erreurs, considérons le programme suivant qui est censé échanger la valeur de deux variables. Il comporte un bogue puisqu'il affiche 2,2 :

```
def exchange(x, y):
    y = x
    x = y
    return x, y

a = 2
b = 3
a, b = exchange(a, b)
print a, b
```

Une stratégie bien connue pour trouver des bugs consiste à écrire vers la console la valeur de certaines variables :

```
def exchange(x, y):
    print "exchange() with params", x, y
```

```
    y = x
    x = y
    print "exchange() returning", x, y
    return x, y

a = 2
b = 3
a, b = exchange(a, b)
print a, b
```

La source de l'erreur devient ainsi évidente et on comprend tout de suite comment corriger l'anomalie.

```
def exchange(x, y):
    print "exchange() with params", x, y
    temp = y
    y = x
    x = temp
    print "exchange() returning", x, y
    return x, y

a = 2
b = 3
a, b = exchange(a, b)
print a, b
```

Maintenant que l'erreur est réglée, les instructions de déboguage sont superflues. Mais au lieu de les supprimer, on peut se contenter de les mettre en commentaires avec le symbole # (Raccourci clavier Ctrl+Q dans TigerJython), ce qui permet éventuellement de les réutiliser par la suite.

```
def exchange(x, y):
#    print "exchange() with params", x, y
    temp = y
    y = x
    x = temp
#    print "exchange() returning", x, y
    return x, y

a = 2
b = 3
a, b = exchange(a, b)
print a, b
```

Il est souvent encore plus adapté d'utiliser un fanion de déboguage permettant d'activer ou de désactiver les informations de déboguage en différents endroits du code :

```
def exchange(x, y):
    if debug: print "exchange() with params", x, y
    temp = y
    y = x
    x = temp
    if debug: print "exchange() returning", x, y
    return x, y

debug = False
a = 2
b = 3
a, b = exchange(a, b)
print a, b
```

En *Python*, il est possible d'échanger les valeurs de deux variables sans créer une autre variable temporaire. Il suffit pour cela d'utiliser sa capacité de faire de l'empaquetage / dépaquetage (*packing / unpacking* en anglais) automatiquement sur les tuples :

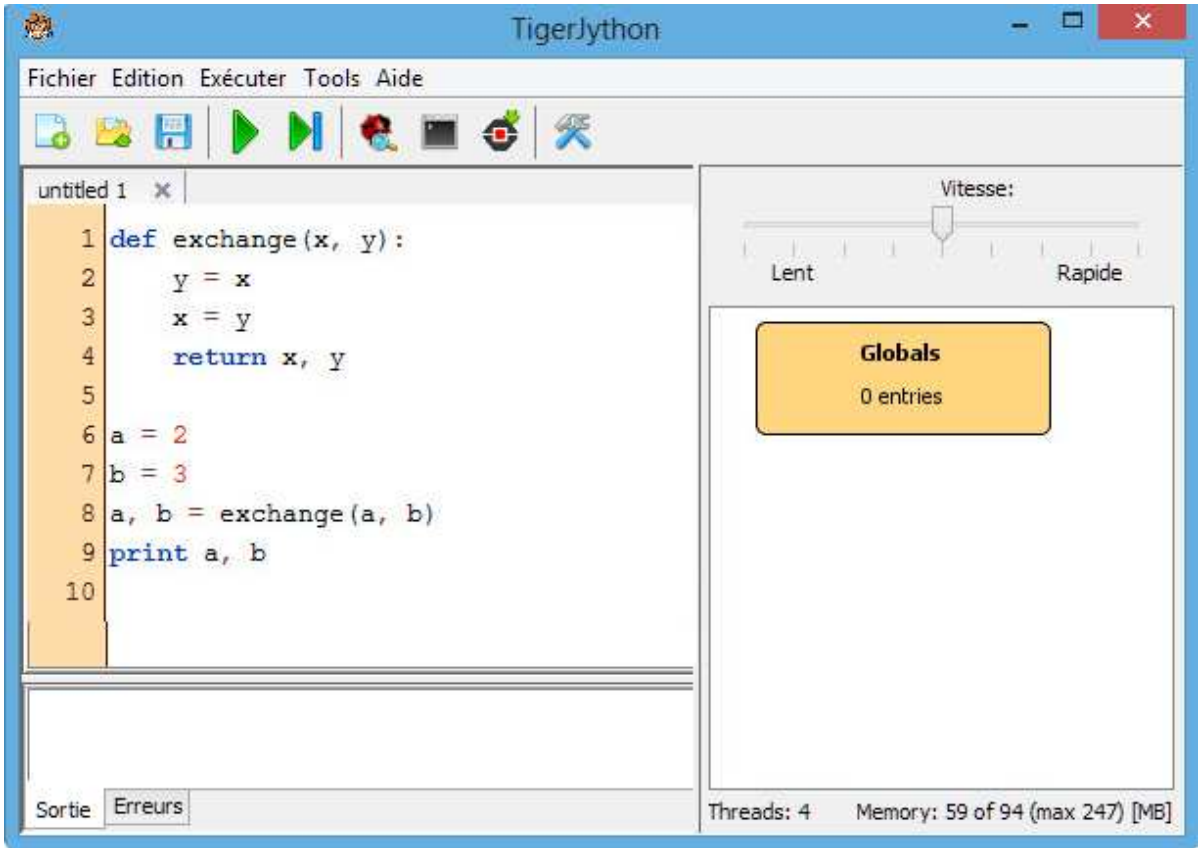
```
def exchange(x, y):
    x, y = y, x
    return x, y

a = 2
b = 3
a, b = exchange(a, b)
print a, b
```

SE SERVIR DU DÉBOGUEUR

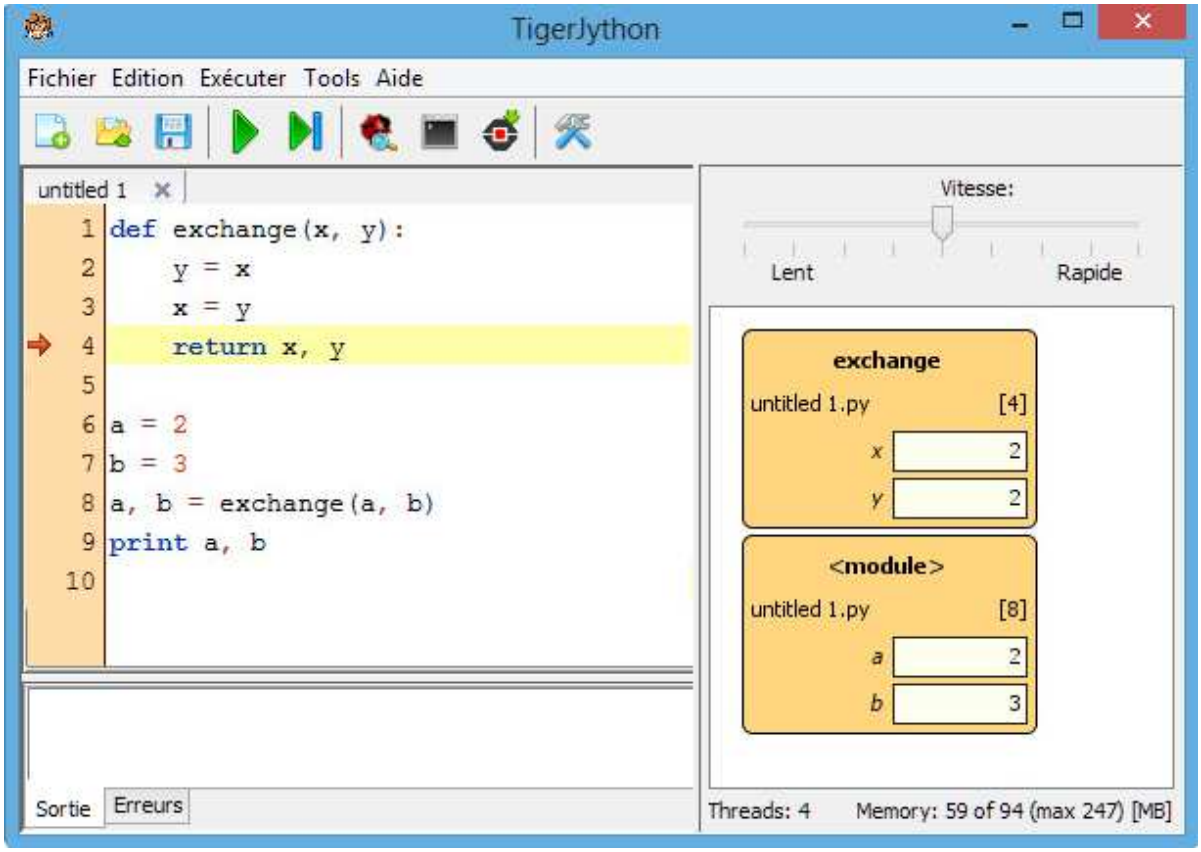
Les débogueurs sont des outils très importants pour développer des gros systèmes logiciels. Ils permettent d'exécuter un programme très lentement ou même pas à pas, ce qui permet de suivre son exécution avec précision. Ils permettent ainsi d'adapter la vitesse d'exécution phénoménale de l'ordinateur aux capacités cognitives et d'analyse limitées de l'être humain. L'éditeur de *TigerJython* intègre un débogueur confortable permettant de faciliter une compréhension précise et détaillée du déroulement de la séquence d'instructions du programme. Nous allons maintenant exécuter le programme précédent en utilisant le débogueur :

Une fois dans l'éditeur, cliquer sur le bouton du débogueur représenté par la coccinelle.

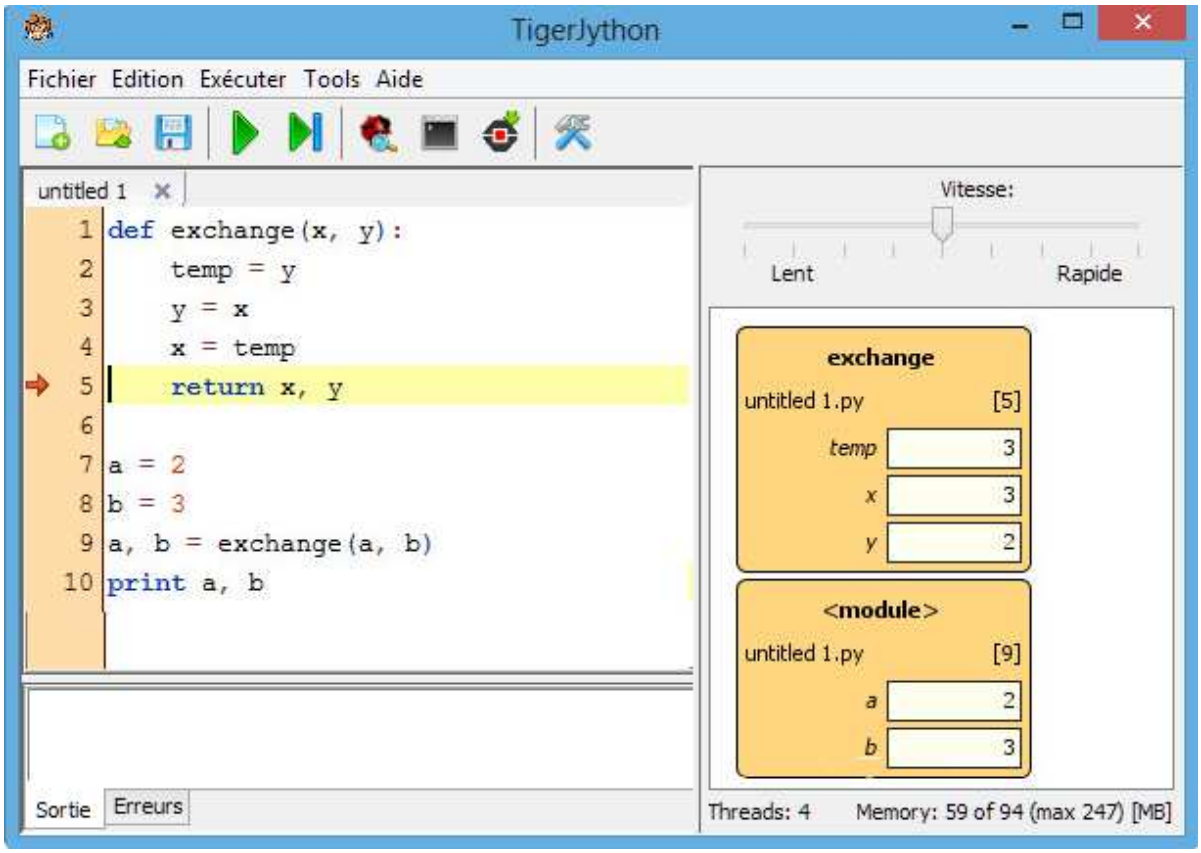
Le bouton portant le « 1 » permet d'exécuter le programme pas à pas en observant l'évolution des différentes variables dans la fenêtre de déboguage. Après 8 clics, on peut voir que les deux variables, *x* et *y*, possèdent la valeur 2 juste avant que la fonction *exchange* ne retourne.



Si l'on exécute le programme une fois le bogue réglé, on peut observer que les valeurs des variables sont effectivement échangées à l'intérieur de la fonction *exchange()*, ce qui mène finalement au résultat attendu. On peut observer également très clairement la durée de vie limitée des variables locales *x* et *y* par opposition aux variables globales *a* et *b*.



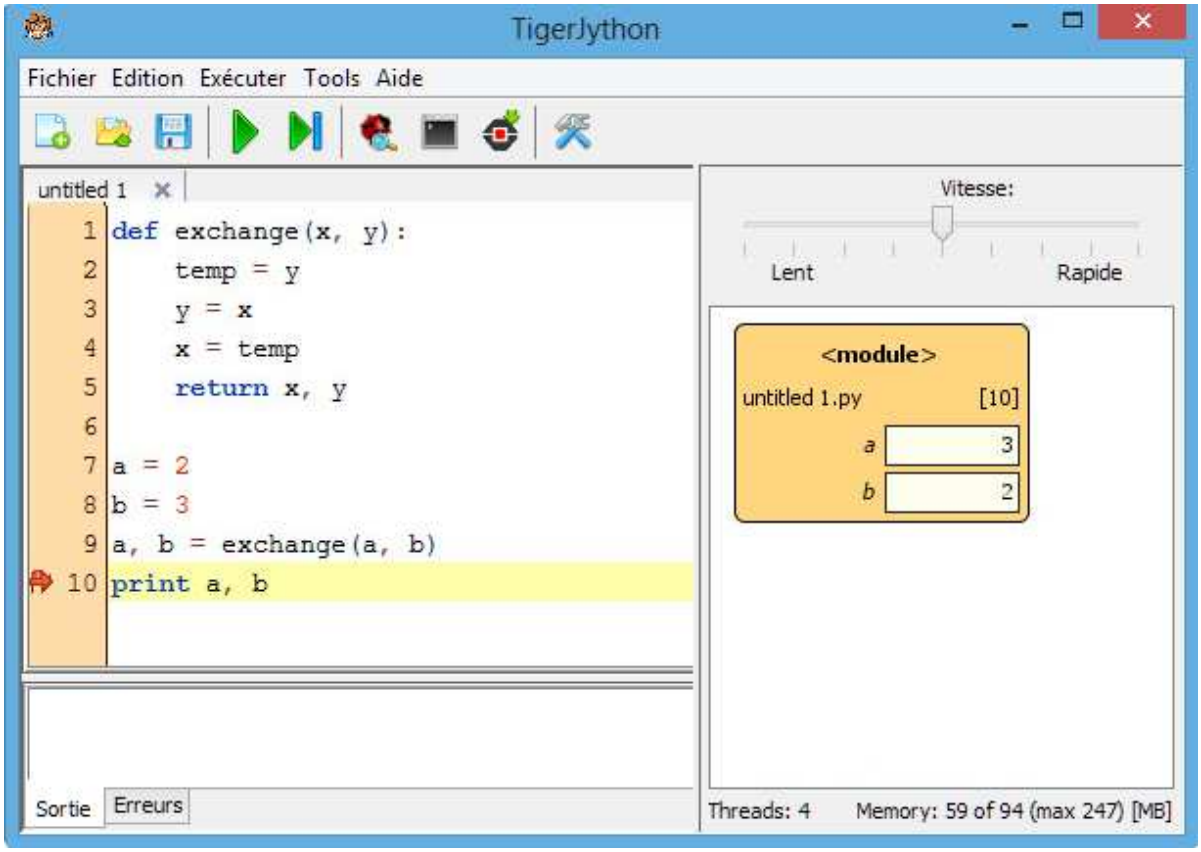
Il est également possible de placer des points d'arrêt dans le programme pour éviter de devoir traverser des parties non pertinentes du code en mode pas à pas. Pour ce faire, cliquer dans la marge gauche, à gauche du numéro de ligne. Un petit fanion rouge représentant le point d'arrêt apparaît alors à cet endroit. Lors d'une pression sur le bouton « exécuter », le programme sera alors exécuté automatiquement et rapidement jusqu'au point d'arrêt et sera ensuite mis en pause.



Ceci permet d'examiner les variables à ce stade de l'exécution du programme. On peut alors

poursuivre l'exécution du programme en mode normal ou en mode pas à pas.

Il est également possible de placer plusieurs points d'arrêt. Pour supprimer un point d'arrêt, il suffit de cliquer dessus.



GESTION DES ERREURS À L'AIDE DES EXCEPTIONS

La gestion des erreurs à l'aide des exceptions est classique dans les langages de programmation modernes. Cela consiste à placer le code susceptible d'engendrer des erreurs dans un bloc *try*. Si l'erreur survient, l'exécution du bloc est abandonnée et c'est le bloc *except* qui alors exécuté. Le bloc *except* permet donc de réagir convenablement à l'erreur. Au pire, on peut y terminer proprement le programme en appelant *sys.exit()*.

On peut par exemple intercepter l'erreur survenant si le paramètre de la fonction *sinc(x)* n'est pas numérique et gérer celle-ci de manière appropriée :

```
from sys import exit
from math import sin

def sinc(x):
    try:
        if x == 0:
            return 1.0
        y = sin(x) / x
    except TypeError:
        print "Error in sinc(x). x =", x, "is not a number"
        exit()
    return y

print sinc("python")
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

La postcondition pour la fonction *sinc(x)* pourrait également être que la fonction retourne *None* si le paramètre est d'un type incorrect. Il revient alors à l'utilisateur de gérer cette erreur convenablement.

```
from math import sin

def sinc(x):
    try:
        if x == 0:
            return 1.0
        y = sin(x) / x
    except TypeError:
        return None
    return y

y = sinc("python")
if y == None:
   print "Illegal call"
else:
   print y
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

11.4 TRAITEMENT PARALLÈLE

INTRODUCTION

D'après le modèle von Neumann, on peut se représenter un ordinateur comme une machine séquentielle qui, d'après un programme, exécute des instructions les unes après les autres occupant chacune le processeur un certain temps. Dans ce modèle, aucune simultanéité n'est possible et, de ce fait, ni traitement parallèle ni concurrence. Dans la vie de tous jours, les processus parallèles sont pourtant omniprésents : chaque être vivant existe en tant qu'individu à part entière mais de nombreux processus se déroulent simultanément à l'intérieur de son corps.

Les avantages du traitement parallèle sont évidents : il permet un gain de performance énorme puisque plusieurs tâches sont résolues en même temps. De plus, la redondance et la chance de survie augmente puisque la panne d'un composant du système n'entraîne pas nécessairement la panne du système dans son ensemble.

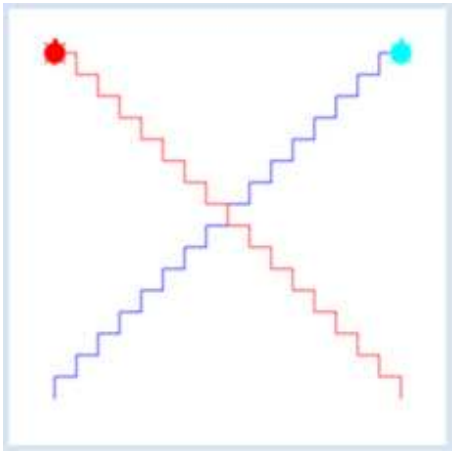
La parallélisation des algorithmes est toutefois une tâche ardue qui, malgré des efforts de recherche considérables, n'est pas encore parvenue à maturité. Le problème essentiel est que les sous-processus partagent généralement des ressources et dépendent de l'issue d'autres processus.

Un **processus léger** ou **fil d'exécution** (*thread* en anglais) est constitué de code s'exécutant en parallèle au sein d'un même programme et un **processus** est constitué de code qui est exécuté en parallèle hors du programme par le système d'exploitation. Python supporte bien les deux types de parallélismes. Dans cette section, nous n'aborderons cependant uniquement l'utilisation de plusieurs threads au sein d'un même processus, à savoir la **programmation multi-thread**.

LE MULTI-THREADING EST PLUS SIMPLE QU'IL N'EN A L'AIR

En *Python*, il est très facile d'exécuter le code d'une fonction dans son propre thread. Il suffit pour cela d'importer le module *threading* et de passer à *start_new_thread()* le nom de la fonction à exécuter en parallèle ainsi que d'éventuels paramètres qu'il faut empaqueter dans un tuple. Le thread débute alors immédiatement son exécution avec le code de la fonction mentionnée..

Notre premier programme faisant usage des threads met en œuvre deux tortues qui dessinent chacune un escalier simultanément et de manière indépendante. Il faut pour cela une fonction nommée de manière arbitraire, en l'occurrence *paint()*, qui peut prendre des paramètres comme la tortue qui doit réaliser l'opération ainsi qu'un fanion indiquant s'il faut effectuer le dessin vers la gauche ou vers la droite. On passe ensuite le nom de cette fonction sans les parenthèses ainsi qu'un tuple contenant les arguments (la tortue et le fanion) à la méthode *thread.start_new_thread()*. Et c'est parti pour notre premier programme parallèle !



```
from gturtle import *
import thread
```

```
def paint(t, isLeft):
    for i in range(16):
        t.forward(20)
        if isLeft:
            t.left(90)
        else:
            t.right(90)
        t.forward(20)
        if isLeft:
            t.right(90)
        else:
            t.left(90)

tf = TurtleFrame()
john = Turtle(tf)
john.setPos(-160, -160)
laura = Turtle(tf)
laura.setColor("red")
laura.setPenColor("red")
laura.setPos(160, -160)
thread.start_new_thread(paint, (john, False))
thread.start_new_thread(paint, (laura, True))
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Pour mouvoir les deux tortues dans la même fenêtre, il faut utiliser explicitement un objet *TurtleFrame tf* et le passer en argument au constructeur des tortues.

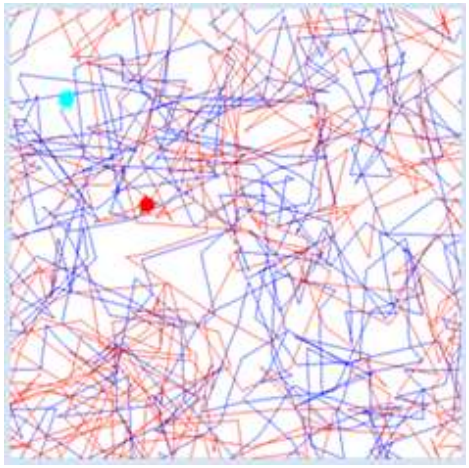
Un nouveau thread est créé et démarré immédiatement avec *start_new_thread()*. Le thread est terminé dès que l'exécution de la fonction avec laquelle il a été lancé parvient à son terme.

La liste de paramètres doit être spécifiée dans un tuple. Notez qu'il faut de ce fait passer un tuple vide *()* pour démarrer une fonction qui ne prend aucun paramètre. Attention encore à la subtilité suivante pour un tuple ne comportant qu'un seul élément *x* : il n'est pas représenté par *(x)* mais par *(x,)*.

CRÉER ET DÉMARRER DES THREADS EN TANT QU'INSTANCE D'UNE CLASSE

On peut obtenir un peu plus de liberté d'action en définissant une classe dérivée de la classe *Thread*. Dans cette classe dérivée on redéfinit la méthode *run()* contenant le code à exécuter.

Pour démarrer un nouveau thread, on commence par créer une instance de cette classe dérivée et l'on appelle sa méthode *start()*. Le système se charge alors d'exécuter la méthode *run()* automatiquement dans un nouveau thread qui sera terminé dès que l'exécution de *run()* touche à sa fin.



```
from threading import Thread
import random
from gturtle import *

class TurtleAnimator(Thread):
    def __init__(self, turtle):
```

```
        Thread.__init__(self)
        self.t = turtle

    def run(self):
        while True:
            self.t.forward(150 * random.random())
            self.t.left(-180 + 360 * random.random())

tf = TurtleFrame()
john = Turtle(tf)
john.wrap()
laura = Turtle(tf)
laura.setColor("red")
laura.setPenColor("red")
laura.wrap()
thread1 = TurtleAnimator(john)
thread2 = TurtleAnimator(laura)
thread1.start()
thread2.start()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Même dans un système multiprocesseur, le code n'est pas vraiment exécuté en parallèle mais plutôt par tranches temporelles successives. De ce fait, le traitement des données demeure généralement "quasi" parallèle. Il est toutefois important de comprendre que l'allocation du temps processeur aux différents threads a lieu à des instants non prévisibles, à savoir n'importe où au milieu de l'exécution d'un bout de code. Lorsque le processeur passe à l'exécution d'un autre thread, le point d'interruption du code du thread actuel ainsi que l'état des variables locales sont bien entendu sauvegardés et restaurés lors de la reprise de l'exécution du thread. Il n'en demeure pas moins que des problèmes épineux peuvent survenir lorsque d'autres threads modifient entre temps des variables globales, ce qui s'applique également au contenu d'une fenêtre graphique. Il ne va donc pas de soi qu'aucun problème ne survienne lorsque l'on a deux tortues dans deux threads différents. [**plus...**].

Comme vous pouvez le constater, la partie principale du programme se termine mais les deux threads continuent d'effectuer leur tâche jusqu'à la fermeture de la fenêtre.

TERMINER LES THREADS

Une fois démarré, un thread ne peut pas être terminé directement depuis une méthode extérieure à lui-même, comme par exemple depuis un autre thread. Pour que le thread se termine, il faut nécessairement que l'exécution de sa méthode *run()* se termine. C'est la raison pour laquelle il n'est jamais une bonne idée de mettre une boucle *while* infinie dans la méthode *run()* d'un thread. Il faut plutôt utiliser une variable globale booléenne *isRunning* pour contrôler l'exécution de la boucle *while*. Cette variable sera normalement mise à *True* tant que le thread doit s'exécuter mais peut également être mise sur *False* depuis un autre thread pour demander son arrêt.

Dans le programme suivant, les deux tortues effectuent un mouvement aléatoire jusqu'à ce que l'une des deux sorte de la surface circulaire.

```
from threading import Thread
import random, time
from gturtle import *

class TurtleAnimator(Thread):
    def __init__(self, turtle):
        Thread.__init__(self)
```

```
        self.t = turtle

    def run(self):
        while isRunning:
            self.t.forward(50 * random.random())
            self.t.left(-180 + 360 * random.random())

tf = TurtleFrame()
john = Turtle(tf)
laura = Turtle(tf)
laura.setColor("red")
laura.setPenColor("red")
laura.setPos(-200, 0)
laura.rightCircle(200)
laura.setPos(0, 0)
thread1 = TurtleAnimator(john)
thread2 = TurtleAnimator(laura)
isRunning = True
thread1.start()
thread2.start()

while isRunning and not tf.isDisposed():
    if laura.distance(0, 0) > 200 or john.distance(0, 0) > 200:
        isRunning = False
    time.sleep(0.001)
tf.setTitle("Limit exceeded")
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

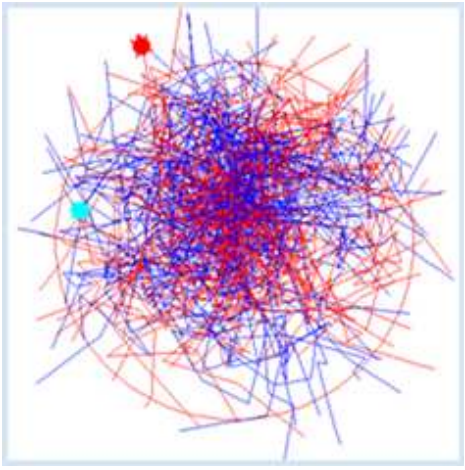
MEMENTO

Il ne faudrait jamais utiliser une boucle « serrée » qui n'effectue aucune action car cela gaspille beaucoup de temps processeur pour rien. Il faut toujours aérer la boucle avec un court temps d'attente avant de boucler en recourant à *time.sleep()*, *Turtle.sleep()* ou *GPanel.delay()*.

Une fois qu'un thread s'est terminé, il n'est plus possible de le redémarrer. Un nouvel appel à la méthode *start()* produit en effet un message d'erreur.

METTRE DES THREADS EN PAUSE

Pour ne suspendre un thread que pour un temps, on peut sauter les instructions présentes dans *run()* en utilisant un fanion global *isPaused* et reprendre leur exécution ultérieurement en mettant *isPaused* sur *False*.



```
from threading import Thread
import random, time
from gturtle import *

class TurtleAnimator(Thread):
    def __init__(self, turtle):
        Thread.__init__(self)
```

```
        self.t = turtle

    def run(self):
        while True:
            if isPaused:
                Turtle.sleep(10)
            else:
                self.t.forward(100 * random.random())
                self.t.left(-180 + 360 * random.random())

tf = TurtleFrame()
john = Turtle(tf)
laura = Turtle(tf)
laura.setColor("red")
laura.setPenColor("red")
laura.setPos(-200, 0)
laura.rightCircle(200)
laura.setPos(0, 0)
thread1 = TurtleAnimator(john)
thread2 = TurtleAnimator(laura)
isPaused = False
thread1.start()
thread2.start()

tf.setTitle("Running")
while not isPaused and not tf.isDisposed():
    if laura.distance(0, 0) > 200 or john.distance(0, 0) > 200:
        isPaused = True
        tf.setTitle("Paused")
        Turtle.sleep(2000)
        laura.home()
        john.home()
        isPaused = False
        tf.setTitle("Running")
    time.sleep(0.001)
```

Programmcode markieren (Ctrl+C pour copier, Ctrl+V pour coller)

Il est même bien plus avisé de stopper un thread avec *Monitor.putSleep()* et de reprendre son exécution par la suite avec *Monitor.wakeUp()*. .

```
from threading import Thread
import random, time
from gturtle import *

class TurtleAnimator(Thread):
    def __init__(self, turtle):
        Thread.__init__(self)
        self.t = turtle

    def run(self):
        while True:
            if isPaused:
                Monitor.putSleep()
            self.t.forward(100 * random.random())
            self.t.left(-180 + 360 * random.random())

tf = TurtleFrame()
john = Turtle(tf)
laura = Turtle(tf)
laura.setColor("red")
laura.setPenColor("red")
laura.setPos(-200, 0)
laura.rightCircle(200)
laura.setPos(0, 0)
thread1 = TurtleAnimator(john)
thread2 = TurtleAnimator(laura)
isPaused = False
thread1.start()
```

```
thread2.start()

tf.setTitle("Running")
while not isPaused and not tf.isDisposed():
    if laura.distance(0, 0) > 200 or john.distance(0, 0) > 200:
        isPaused = True
        tf.setTitle("Paused")
        Turtle.sleep(2000)
        laura.home()
        john.home()
        isPaused = False
        Monitor.wakeUp()
        tf.setTitle("Running")
    time.sleep(0.001)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Un thread peut se mettre lui-même en pause avec la méthode bloquante *Monitor.putSleep()* de sorte qu'il ne gaspille aucun temps processeur. Un autre thread peut alors le réactiver avec la méthode *Monitor.wakeUp()* qui va obliger la méthode *Monitor.putSleep()* à retourner.

ATTENDRE LE RÉSULTAT DU TRAITEMENT D'UN THREAD

Dans ce programme, on utilise un thread ouvrier (*worker thread*) pour calculer la somme des nombres entiers entre 1 et 1'000'000. Le programme principal attend le résultat et affiche le temps nécessaire pour l'effectuer. Comme ce temps peut légèrement varier, on refait la mesure 10 fois de suite dans un thread ouvrier. On utilise *join()* pour attendre la fin de l'exécution du thread.

```
from threading import Thread
import time

class WorkerThread(Thread):
     def __init__(self, begin, end):
         Thread.__init__(self)
         self.begin = begin
         self.end = end
         self.total = 0

     def run(self):
         for i in range(self.begin, self.end):
             self.total += i

startTime = time.clock()
repeat 10:
    thread = WorkerThread(0, 1000000)
    thread.start()
    thread.join()
    print thread.total
print "Time elapsed:", time.clock() - startTime, "s"
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

Comme dans la vie réelle, on peut déléguer du travail pénible à plusieurs ouvriers différents. Si l'on utilise deux threads ouvriers pour que chacun effectue la moitié du travail, il faudra attendre que les deux aient terminé pour combiner leur résultat.

```
from threading import Thread
import time
```

```
class WorkerThread(Thread):
     def __init__(self, begin, end):
         Thread.__init__(self)
         self.begin = begin
         self.end = end
         self.total = 0

     def run(self):
         for i in range(self.begin, self.end):
             self.total += i

startTime = time.clock()
repeat 10:
    thread1 = WorkerThread(0, 500000)
    thread2 = WorkerThread(500000, 1000000)
    thread1.start()
    thread2.start()
    thread1.join()
    thread2.join()
    result = thread1.total + thread2.total
    print result
print "Time elapsed:", time.clock() - startTime, "s"
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

On pourrait également mettre un fanion global *isFinished() à True* lorsque le thread se termine et tester ce fanion dans une boucle *while* dans la partie principale du programme. Cette solution est cependant moins élégante que l'utilisation de *join()* qui présente l'avantage de ne pas gaspiller inutilement des cycles processeur en testant sans arrêt le fanion.

SECTIONS CRITIQUES ET VERROUS

Puisque les threads s'exécutent de manière indépendante, la modification de données communes par plusieurs threads est délicate. Pour éviter des collisions non souhaitées entre les threads, on regroupe les instructions qui doivent nécessairement être exécutées sans interruption en les regroupant dans un bloc de programme appelée *section critique* muni de protections garantissant qu'il sera exécuté sans interruption, de manière atomique. Si un autre thread tente d'exécuter ce bloc de code, il doit attendre jusqu'à ce que le thread actuel ait terminé l'exécution de ce même bloc de code. Cette protection est implémentée dans les programmes Python à l'aide d'un **verrou** (*lock* en anglais). Un verrou est une instance de la classe *Lock* qui possède deux états possibles : verrouillé (*locked*) et déverrouillé (*unlocked*) ainsi que deux méthodes, *acquire()* et *release()* fonctionnant selon les règles suivantes :

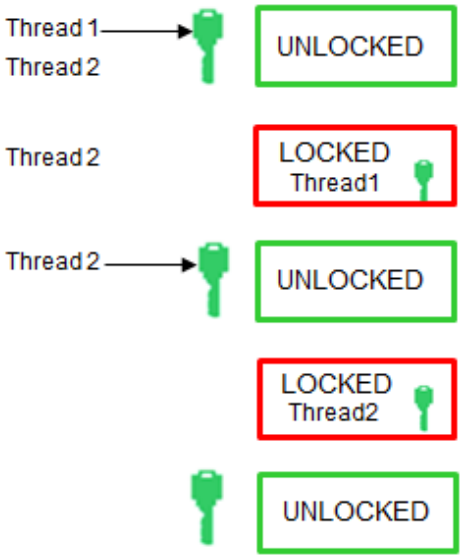
état	**appel**	**état / activité subséquent**
déverrouillé	acquire()	verrouillé
verrouillé	acquire()	verrouillé jusqu'à ce qu'un autre thread appelle *release()*
déverrouillé	release()	Message d'erreur (RuntimeException)
verrouillé	release()	déverrouillé

On dit qu'un thread acquière le verrou avec *acquire()* et le libère à nouveau avec *release()* [**plus...**].

Voici plus précisément comment procéder pour protéger une section critique : on commence par créer un objet global *lock = Lock()* qui est bien entendu dans un état initial déverrouillé. Chaque thread tente ensuite d'acquérir le verrou avec *acquire()* en entrant dans la section critique. En

cas d'échec, si le verrou est déjà verrouillé par un autre thread, le thread demandeur est automatiquement mis en veille jusqu'à la libération du verrou. Lorsqu'un thread acquière le verrou, il traverse la section critique et libère le verrou lorsqu'il a terminé avec *release()* de sorte que les autres threads puissent à leur tour l'acquérir [**plus...**].

On peut comparer une section critique dans le code comme une ressource contenue dans une pièce munie d'une porte fermée à clé. Le verrou de la section critique agit alors comme une clé suspendue à l'entrée de la pièce nécessaire à un thread pour entrer dans la pièce. Un thread, lorsqu'il pénètre dans la pièce (section critique) prend alors la clé avec lui et referme la porte derrière lui. Tous les threads qui voudraient pénétrer dans la pièce doivent alors faire la queue pour attendre la clé. Une fois que le thread actuellement dans la pièce a terminé son travail, il ressort de la pièce et suspend la clé à son emplacement. Le premier thread qui attend devant la porte peut alors pénétrer la pièce et effectuer son travail. Si aucun thread n'attend devant la porte, la clé reste simplement suspendue jusqu'à ce qu'un prochain thread tente d'entrer dans la section critique.



Dans le programme suivant, le dessin et la suppression de carrés remplis constitue la section critique. On efface un carré en repeignant par-dessus avec la couleur d'arrière-fond blanche. Le thread principal crée un carré rouge clignotant en dessinant le carré rouge et en l'effaçant après un court laps de temps. Dans un second thread *MyThread*, le clavier est sans arrêt interrogé avec *getKeyCode()*. Si l'utilisateur presse sur la barre d'espace, le carré est déplacé à une position aléatoire.

Il est évident que la section critique doit être protégée par un verrou. Si le déplacement du carré avait lieu pendant qu'il en train d'être dessiné ou effacé, le comportement serait chaotique.

```
from gpanel import *
from threading import Thread, Lock
import random

class MyThread(Thread):
    def run(self):
        while not isDisposed():
            if getKeyCode() == 32:
                print "----------- Lock requested by MyThread"
                lock.acquire()
                print "----------- Lock acquired by MyThread"
                move(random.randint(2, 8), random.randint(2, 8))
                delay(500)  # for demonstration purposes
                print "----------- Lock releasing by MyThread..."
                lock.release()
            else:
                delay(1)

def square():
    print "Lock requested by main"
    lock.acquire()
    print "Lock acquired by main"
    setColor("red")
    fillRectangle(2, 2)
    delay(1000)
    setColor("white")
    fillRectangle(2, 2)
    delay(1000)
```

```
    print "Lock releasing by main..."
    lock.release()

lock = Lock()
makeGPanel(0, 10, 0, 10)

t = MyThread()
t.start()
move(5, 5)
while not isDisposed():
    square()
    delay(1) # Give up thread for a short while
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

On peut observer dans la console la manière dont chaque thread attend sagement son tour jusqu'à ce que le verrou soit libéré :

```
Lock requested by main
Lock acquired by main
---------- Lock requested by MyThread
Lock releasing by main...
---------- Lock acquired by MyThread
Lock requested by main
---------- Lock releasing by MyThread...
Lock acquired by main
```

Essayez de désactiver le mécanisme du verrou en commentant les lignes où il est acquis et libéré. Vous verrez que le carré n'est plus dessiné et effacé correctement.

Rappelez-vous également qu'il faut toujours ajouter un petit temps d'attente dans la boucle *while* pour éviter de consommer trop de cycles processeur inutilement.

GUI-WORKERS

Les fonctions de rappel lancées par un composant GUI (*GUI = Graphical User Interface*) s'exécutent dans un thread propre au système nommé *Event Dispatch Thread (EDT)*. Celui-ci est responsable de garantir un rendu correct de la fenêtre graphique ainsi que de ses différents composants (boutons, etc.). Du fait que le rendu s'effectue à la fin de la fonction de rappel, l'interface graphique est gelée jusqu'à ce que cette dernière se termine. Il n'est de ce fait pas possible de programmer des animations graphiques au sein des fonctions de rappel d'interface graphique. Il faut impérativement suivre la règle suivante de manière très stricte :

Les fonctions de rappel (callbacks) d'interface graphique (GUI) doivent retourner très rapidement et n'impliquer aucun traitement de longue durée (> 10 ms).

En l'occurrence, une tâche est considérée de longue durée si son exécution excède une durée de 10 ms. Il faut estimer cette durée d'exécution en tenant compte des pires conditions possibles telles qu'un processeur lent et une charge système élevée. Si une action dure trop longtemps, il faut l'exécuter dans un thread séparé appelé **GUI worker** en anglais.

Le programme suivant dessine une rosace lors d'un clic sur l'un des deux boutons. Le dessin est animé et prend un certain temps. Il faut de ce fait effectuer le dessin dans un thread ouvrier, ce qui ne devrait vous poser aucun problème vu vos connaissances sur les threads.

Ce fonctionnement soulève cependant un autre problème : puisque chaque clic sur un bouton va lancer un nouveau thread de rendu, plusieurs dessins peuvent être démarrés simultanément, ce

qui va rapidement conduire au chaos le plus absolu. On peut éviter cette situation très simplement en désactivant les boutons durant l'exécution du dessin [**plus...**].

```
from gpanel import *
from javax.swing import *
import math
import thread

def rho(phi):
    return math.sin(n * phi)

def onButtonClick(e):
    global n
    enableGui(False)
    if e.getSource() == btn1:
        n = math.e
    elif e.getSource() == btn2:
        n = math.pi
#    drawRhodonea()
    thread.start_new_thread(drawRhodonea, ())

def drawRhodonea():
    clear()
    phi = 0
    while phi < nbTurns * math.pi:
        r = rho(phi)
        x = r * math.cos(phi)
        y = r * math.sin(phi)
        if phi == 0:
          move(x, y)
        else:
          draw(x, y)
        phi += dphi
    enableGui(True)

def enableGui(enable):
    btn1.setEnabled(enable)
    btn2.setEnabled(enable)

dphi = 0.01
nbTurns = 100
makeGPanel(-1.2, 1.2, -1.2, 1.2)
btn1 = JButton("Go (e)", actionListener = onButtonClick)
btn2 = JButton("Go (pi)", actionListener = onButtonClick)
addComponent(btn1)
addComponent(btn2)
validate()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Seul un code de très courte durée (< 10 ms) peut être placé dans les fonctions de rappel lancées par les composants GUI. Placer un code prenant plus de temps dans un gestionnaire d'événement de l'interface graphique mène au gel de l'interface utilisateur, ce qui est très désagréable. Il faut donc déléguer les traitements trop longs (> 10 ms) à un thread ouvrier séparé.

Dans une interface graphique, seuls les boutons ou menus menant à des actions permises et utiles devraient être actifs. Les composants qui lancent des opérations interdites ou insensées devraient être désactivés.

SITUATIONS DE COMPÉTITION, INTERBLOCAGE

Les êtres humains travaillent de manière hautement parallèle mais leur raisonnement logique est essentiellement séquentiel. Il nous est de ce fait très difficile de conserver une vue d'ensemble de l'exécution de programmes multi threads. Voilà pourquoi il faut toujours utiliser les threads avec une grande précaution, aussi élégants puissent-ils paraître de prime abord.

Hormis dans les applications basées sur des données et traitements aléatoires, un programme devrait toujours retourner le même résultat (postcondition) lorsqu'on lui fournit les mêmes conditions initiales (préconditions). Ceci n'est en aucun cas garanti pour les programmes comportant plusieurs threads accédant à des données partagées, même si les sections critiques sont protégées par des verrous. Le programme suivant démontre cette affirmation : il comporte deux threads, *thread1* et *thread2*, effectuant une addition et une multiplication de deux nombres globaux *a* and *b*. Les variables globales *a* et *b* sont protégées par les verrous *lock_a* et *lock_b*. La programme principal initialise et démarre les deux threads et attend ensuite qu'ils se terminent. La valeur finale des variables *a* et *b* est finalement affichée dans la console.

Les threads sont ici générés de manière différente en spécifiant la méthode *run()* en tant que paramètre nommé du constructeur de la classe *Thread*.

```
from threading import Thread, Lock
from time import sleep

def run1():
    global a, b
    print "----------- lock_a requested by thread1"
    lock_a.acquire()
    print "----------- lock_a acquired by thread1"
    a += 5
#    sleep(1)
    print "----------- lock_b requested by thread1"
    lock_b.acquire()
    print "----------- lock_b acquired by thread1"
    b += 7
    print "----------- lock_a releasing by thread1"
    lock_a.release()
    print "----------- lock_b releasing by thread1"
    lock_b.release()

def run2():
    global a, b
    print "lock_b requested by thread2"
    lock_b.acquire()
    print "lock_b acquired by thread2"
    b *= 3
#    sleep(1)
    print "lock_a requested by thread2"
    lock_a.acquire()
    print "lock_a acquired by thread2"
    a *= 2
    print "lock_b releasing by thread2"
    lock_b.release()
    print "lock_a releasing by thread2"
    lock_a.release()

a = 100
b = 200
lock_a = Lock()
lock_b = Lock()

thread1 = Thread(target = run1)
thread1.start()
thread2 = Thread(target = run2)
```

```
thread2.start()
thread1.join()
thread2.join()
print "Result: a =", a, ", b =", b
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

En exécutant ce programme suffisamment de fois, on peut observer que le résultat affiché est tantôt a = 205, b = 607, tantôt a = 210, b = 621. Il arrive même parfois que l'exécution du programme ne se termine pas ! Voici l'explication de ce phénomène :

Bien que *thread1* soit créé en premier et démarré depuis le programme principal avant *thread2*, on ne peut pas garantir qu'il soit le premier à pénétrer dans la section critique puisque la première ligne est parfois

```
lock_a requested by thread1
```

et parfois

```
lock_b requested by thread2
```

Le cours des événements n'est ensuite pas non plus uniquement déterminé puisque le basculement entre les threads peut survenir à n'importe quel endroit du code. Il est tout-à-fait possible que la multiplication des nombres *a* et *b* soit effectuée en premier, ce qui explique les résultats divergents. Puisque les deux threads s'exécutent (*run* en anglais) ensemble comme dans une compétition, on parle dans ce cas de **situation de compétition** (*race condition* en anglais)..

La situation peut d'ailleurs prendre une tournure encore plus fâcheuse et geler complètement le programme. Juste avant le blocage du programme, on peut alors observer les lignes suivantes dans la console :

```
---------- lock_a requested by thread1
lock_b requested by thread2
lock_b acquired by thread2
lock_a requested by thread2
---------- lock_a acquired by thread1
---------- lock_b requested by thread1
```

Il faut un sacré travail de détective pour comprendre ce qui s'est passé dans ce cas. Tentons tout de même le coup : apparemment, le *thread1* commence l'exécution en premier et tente d'acquérir le verrou *lock_a*. Avant qu'il n'ait l'occasion de terminer l'exécution de cette section critique, le *thread2* tente d'acquérir le verrou *lock_b* avec succès. Immédiatement, le *thread2* tente également d'acquérir le verrour *lock_a*, mais en vain puisque celui-ci a déjà été acquis par le *thread1*. De ce fait, le *thread2* est bloqué en attendant la libération du verrou par *thread1* qui continue son exécution et tente d'acquérir le verrou *lock_b*. C'est là que les choses se gâtent car le verrou *lock_b est déjà acquis par le thread2* qui est bloqué dans son exécution. Ainsi, les deux threads se bloquent mutuellement sans espoir de pouvoir libérer le verrou nécessaire à l'autre thread pour être débloqué, ce qui bloque tout le programme. On appelle cela une **situation d'interblocage *(deadlock en anglais)***. L'activation des instructions *sleep(1)* par suppression du commentaire permet de conduire le programme systématiquement dans cette condition. Réfléchissez à cela pour essayer de bien comprendre ce phénomène.

Comme vous pouvez le constater, un interblocage survient lorsque deux threads *thread1* et *thread2* dépendent de deux ressources partagées *a* et *b* et se bloquent mutuellement. Par conséquent, il peut arriver que le *thread2* attende sur le verrou *lock_a* et que le *thread1* attende sur le verrou *lock_b* , se bloquant mutuellement de telle sorte que les verrous n'ont aucune chance d'être libérés.

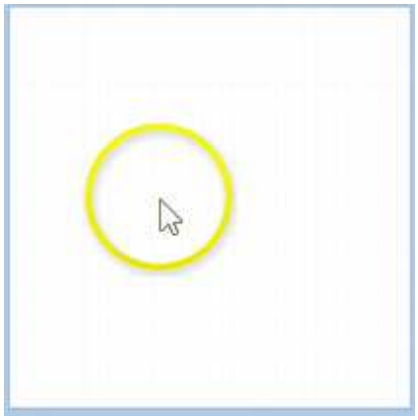
Pour éviter les situations d'interblocage, il faut scrupuleusement adhérer à la règle suivante :

Les ressources partagées devraient être si possible protégées par un seul verrou. De plus, il faut impérativement s'assurer que le verrou soit libéré.

EXPRESSIONS ATOMIQUES ET THREAD-SAFE

Si plusieurs threads sont impliqués dans un programme, on ne connait jamais, en tant que programmeur, à quel moment et à quel endroit du code va avoir lieu le basculement entre threads. Comme nous l'avons déjà constaté, cela peut conduire à des résultats inattendus et incorrects lorsque les threads travaillent avec des ressources partagées. Ceci est particulièrement vrai si plusieurs threads changent une fenêtre de sorte que, lorsqu'un thread ouvrier est généré au sein d'une fonction de rappel GUI pour exécuter un code de longue durée, il faut presque toujours s'attendre à des situations chaotiques. Dans le programme précédent, nous avons évité ce problème en désactivant les boutons durant l'exécution des fonctions de rappel. On peut s'assurer que plusieurs threads puissent s'exécuter en parallèle sans se marcher sur les pieds en prenant des précautions particulières. Un tel code est alors appelé **thread-safe**. Écrire du code thread-safe pour qu'il puisse être exécuté dans en environnement multi-threads sans conflit est un véritable art. [**plus..**].

Il existe peu de bibliothèques thread-safe puisqu'elles sont généralement moins performantes et qu'elles comportent un risque d'interblocage. Comme vous avez pu le constater, la bibliothèque *GPanel* n'est pas thread-safe alors que la bibliothèque de tortues *gTurtle* est thread-safe. On peut en effet déplacer plusieurs tortues dans plusieurs threads de manière quasi simultanée. Dans le programme suivant, chaque clic de souris engendre à la position du clic une nouvelle tortue pilotée par un thread séparé. Chaque tortue dessine et remplit une étoile de manière autonome.



```
from gturtle import *
import thread

def onMousePressed(event):
#     createStar(event)
    thread.start_new_thread(createStar, (event,))

def createStar(event):
    t = Turtle(tf)
    x = t.toTurtleX(event.getX())
    y = t.toTurtleY(event.getY())
    t.setPos(x, y)
    t.startPath()
    repeat 9:
        t.forward(100)
        t.right(160)
    t.fillPath()

tf = TurtleFrame(mousePressed = onMousePressed)
tf.setTitle("Klick To Create A Working Turtle")
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Si l'on ne génère pas un nouveau thread (décommenter la ligne commentée et commenter l'autre) on n'observe les étoiles que lorsqu'elles sont terminées. Il est cependant possible d'écrire le programme sans qu'il n'utilise son propre thread séparé en utilisant le paramètre nommé *mouseHit* au lieu de *mousePressed* comme nous l'avions fait au chapitre **chapitre 2.11**. Dans ce cas, le thread est alors automatiquement engendré par la bibliothèque de graphiques tortues.

Il est important de savoir que le basculement entre threads peut même survenir en plein milieu d'une ligne de code. Un changement de thread peut par exemple survenir au beau milieu de la ligne

a = a + 1

ou même

a += 1

entre le moment où la variable est lue et celui où la valeur y est écrite.

Par contraste, une expression est appelée **atomique** si elle ne peut pas être interrompue par un changement de thread. Comme pour la plupart des langages de programmation, Python n'est pas vraiment atomique. Il peut arriver qu'une instruction *print* soit interrompue par des *print*s survenant dans d'autres threads, ce qui peut résulter en une sortie complètement chaotique vers la console. C'est au programmeur qu'incombe la lourde tâche de rendre atomiques et thread-safe les fonctions, les expressions et les sections du code en recourant aux verrous.

EXERCICES

1. Modifier le programme qui effectue les additions et les multiplications sur les variables globales *a* et *b* de sorte qu'il n'utilise qu'un unique verrou et faire en sorte qu'il ne survienne ni situation de compétition ni interblocage.

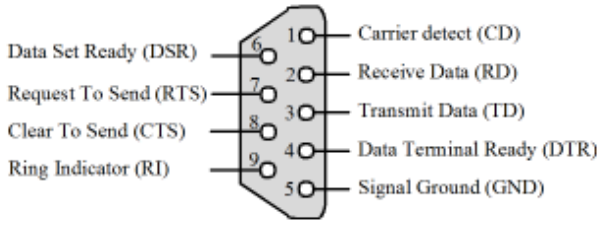
11.5 INTERFACE SÉRIE

INTRODUCTION

Bien que les interfaces Bluetooth, Ethernet et USB soient communément utilisées pour la communication entre un ordinateur et ses périphériques, la communication par interface série (RS-232C) est toujours largement utilisée puisqu'elle nécessite des circuits de contrôle bien moins complexes dans les appareils. C'est la raison pour laquelle les interfaces séries sont encore très utilisées pour connecter des appareils de mesure (voltmètres, oscilloscopes, etc…) aux appareils de contrôle et aux robots ou pour communiquer avec des microcontrôleurs. Les ordinateurs modernes n'ont plus de port série mais il est très facile de se procurer à bon prix un adaptateur USB vers port série.

Pour comprendre les interfaces série, il faut comprendre qu'elles comportent des lignes pour envoyer et recevoir des données (TD/TR), deux paires de lignes de poignée de main (*handhake* line en anglais) RTS/CTS et DTR/DSR, deux lignes d'état (*status* en anglais) CD/RI et une ligne de masse (*ground* en anglais). On peut observer les lignes de sortie TD, RTS, DTR et les lignes d'entrée RD, CTS, DSR, CD, RI présentes sur le port d'un vieil ordinateur. Les lignes RTS et DTR peuvent être activées et désactivées par le programme et les lignes CTS, DSR, CD et RI ne sont accessibles qu'en lecture.

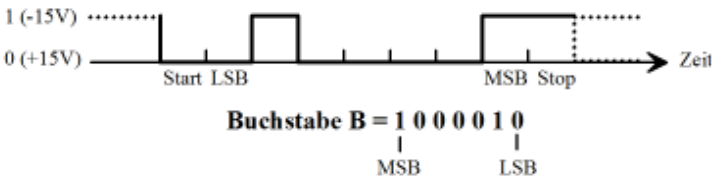
Connexions du connecteur 9-pin RS-232 :



Le format des données transmises est simple. Il est constitué d'octets de données transmis chronologiquement en série (les uns après les autres). Le transfert débute par un bit de démarrage (start bit) qui permet au périphérique en réception de prêter attention aux données qui vont être transférées. Les données à proprement parler suivent ensuite en 5, 6, 7 ou, le plus souvent, 8 bits. Afin de faciliter la correction d'erreur, ces bits sont généralement suivis d'un bit de parité qui indique si un nombre pair ou impair de bits de données à 1 ont été envoyés. Ce bit n'est cependant pas obligatoire et peut être omis. Le transfert est terminé par un ou deux bits d'arrêt (*stop bit*). Les appareils émetteur et en réception ne sont pas synchronisés l'un avec l'autre : le transfère peut débuter et se terminer à n'importe quel moment. Il est cependant nécessaire que les deux appareils se mettent d'accord sur la durée d'un seul bit. Cette valeur est spécifiée en bauds (bit / seconde) et est généralement restreinte à l'une des valeurs standard suivantes : 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200 bauds. De plus, les deux appareils peuvent se mettre d'accord sur une poignée de mains (contrôle de flux) leur permettant de s'informer s'ils sont prêts pour le transfert. On peut distinguer entre les poignées de mains matérielles (utilisation des lignes de poignées de mains de l'interface série) et logicielles (insertion de caractères ASCII spéciaux (XON/XOFF) dans le flux de données.

Une configuration de port série typique est donc constituée des informations suivantes : le débit en bauds, le nombre de données à transmettre, le nombre de bits d'arrêts, le bit de parité (aucune, pair, impair), la poignée de mains (aucune, matérielle, logicielle).

Voici comment se présenterait un diagramme temporel du voltage sur la ligne de données pour transmettre la lettre 'B' avec la configuration *7* bits de données/pas de parité/1 bit d'arrêt.



INSTALLATION

Le module *pySerial* décrit dans cette section fonctionne sur un système 32 ou 64-bit, mais uniquement sur un Java Runtime Environment (JRE) 32 bits. Il faut effectuer les opérations supplémentaires décrites ci-après (*Lib* est un sous-dossier du dossier dans lequel l'archive *tigerjython2.jar* est située):

1. **Télécharger** *pyserial.zip* décompresser l'archive puis copier l'intégralité de son contenu dans le dossier *Lib.*

```
<tigerjythonhome>
   Lib
      Serial
         tools
         urhandler
```

2. **Télécharger** *javacomm.zip*. Décompresser l'archive et copier *comm.jar* dans le dossier *Lib*.

3. (Sur Windows): copier le fichier *win32com.dll* présent dans l'archive *javacomm.zip* dans le dossier *c.\windows\system32*. Dans *c:\Program Files (x86)* se trouve le dossier *Java* et le répertoire de base de la JRE. Copier le fichier *javax.com.properties* présent dans le fichier *javacomm.zip* vers le sous-dossier *Lib*. (Remarque : Lors d'une mise à jour de la JRE, il peut être nécessaire de recopier ce fichier.)

4. Si nécessaire, installer le driver de l'adaptateur USB et connecter l'adaptateur au port USB. Chercher ou réinitialiser le port COM attribué dans la section « Ports (COM et LPT) » du gestionnaire de périphériques de Windows, par exemple COM1 ou COM3.

5. Tester l'installation en exécutant (double clic) l'archive JAR *PortEnumerator.jar* (présente dans *javacomm.zip*). Le port COM utilisé devrait être affiché.

SIMPLE TERMINAL

Pour le moment, il vous faudra vous contenter de la documentation de *pySerial*. Sur la page **https://github.com/pyserial/pyserial** sous *pySerial API,* on trouve une description complète des classes nécessaires. Certaines d'entre elles sont cependant dépendantes de la plateforme.

Le programme suivant permet d'envoyer à un périphérique externe les caractères saisis au clavier les uns après les autres et d'afficher dans la console les caractères reçus. C'est la forme la plus simple d'un émulateur de terminal. Pour le tester, deux options se présentent. La première consiste à connecter à l'autre extrémité un ordinateur configuré de manière appropriée sur lequel vous exécutez également le même programme. Pour que cela fonctionne, il faut utiliser un câble série spécial qui croise les lignes RD (Receive Data) et TD (Transmit Data). La deuxième possibilité consiste à relier les pins RD et TD de votre port série par un fil ou un autre objet conducteur, ce qui aura pour effet d'afficher dans la console les caractères saisis puisque les données envoyées en sortie de l'interface série seront immédiatement retournées en entrée.

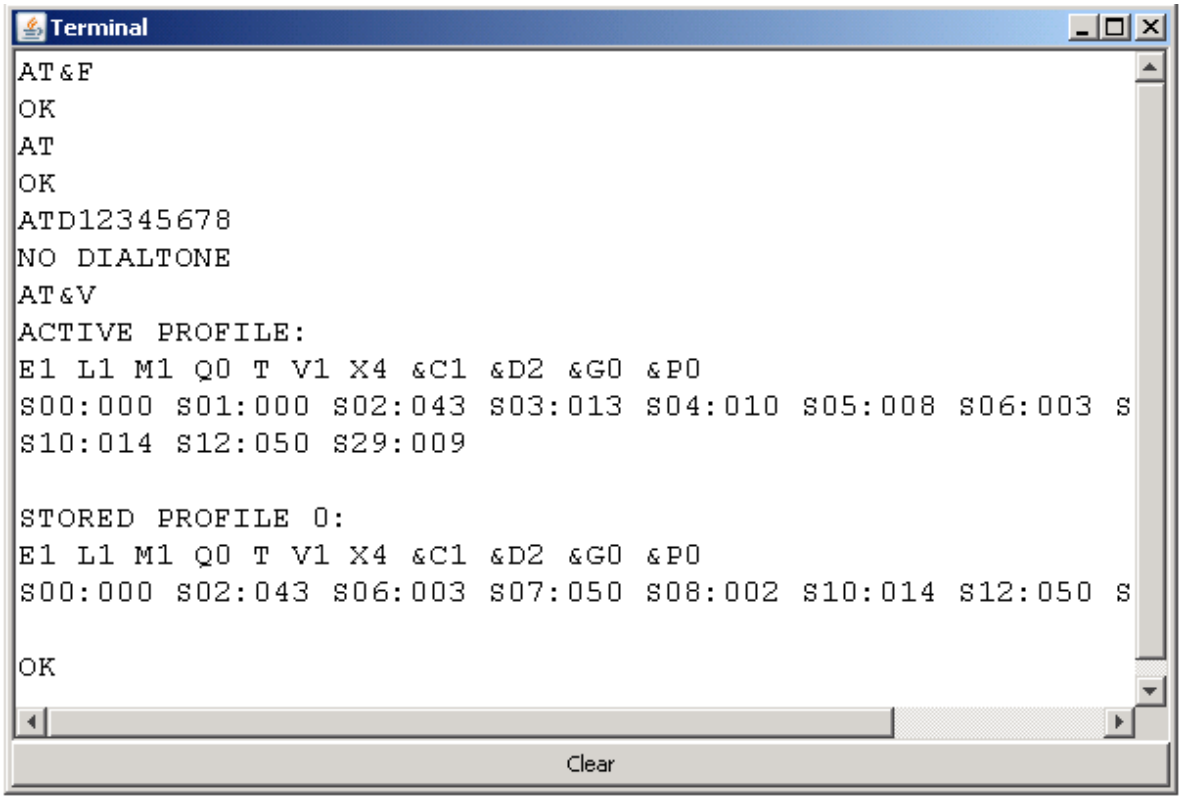
```
import serial
from gconsole import *

makeConsole()
setTitle("Terminal")
ser = serial.Serial(port = "COM1", baudrate = 2400, timeout = 0)
while not isDisposed():
    delay(1)
    ch = getKey()
    if ch != KeyEvent.CHAR_UNDEFINED:  # a key is typed
        ser.write(ch)
    nbChars = ser.inWaiting()
    if nbChars > 0:
        text = ser.read(nbChars)
        for ch in text:
            if ch == '\n':
                gprintln()
            else:
                gprint(ch)
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

MEMENTO

Pour lire les caractères reçus en entrée de la ligne, il faut utiliser une fonction non bloquante puisque le programme doit sans arrêt vérifier si une touche du clavier a été pressée. La méthode *ser.read()* est non bloquante si le paramètre de timeout du constructeur et mis à 0.

Si vous avez un notebook disposant d'un modem intégré, le programme de terminal peut l'utiliser pour communiquer avec le jeu de commandes Hayes comme le montre la figure ci-dessous.



11.6. SOCKETS TCP

INTRODUCTION

L'échange de données entre ordinateurs joue un rôle extrêmement important dans notre monde connecté. On parle donc souvent des technologies de l'information et de la communication qui devraient être maitrisées par tous. Dans ce chapitre, nous allons apprendre à gérer les échanges de données entre deux systèmes informatiques en utilisant le protocole TCP/IP qui constitue la base de toutes les connexions Internet comme le Web et les services des streaming (données dans le nuage, transmission de la voix, de musique, de vidéo).

TCPCOM: UNE BIBLIOTHÈQUE SOCKET ORIENTÉE ÉVÉNEMENTS

La programmation socket est basée sur le modèle client-serveur que nous avons déjà traité dans la **section 6.2**. Le rôle du serveur et du client ne sont pas complètement symétriques. En effet, le serveur doit être démarré en premier avant qu'un client ne puisse s'y connecter. Pour identifier le serveur sur l'Internet, on utilise son adresse IP. De plus, le serveur dispose de 65536 canaux de communication (ports IP) qui sont désignés par un nombre compris entre 0 et 64535.

Lorsque le serveur démarre, il crée un socket serveur utilisant un port bien défini et passant ensuite en attente. On pourrait comparer un socket réseau à une prise murale à laquelle on peut « brancher » une connexion. On dit que le serveur écoute sur le port : il est donc en état d'attente de la connexion d'un client. Pour se connecter au serveur, le client crée un socket client semblable à une fiche électrique et tente d'établir un lien de communication avec le serveur en utilisant l'adresse et le port IP appropriés.

La bibliothèque *tcpcom* simplifie la programmation socket de manière drastique puisqu'elle décrit l'état courant du serveur et du client à l'aide de variables d'état. Le changement d'état est causé par un événement. Ce modèle de programmation correspond parfaitement à l'idée que l'on se fait naturellement de la communication entre deux partenaires comme une suite d'événements chronologiques.

Comme d'ordinaire dans le modèle événementiel, une fonction de rappel (callback), en l'occurrence appelée *stateChanged(state, msg)*, est invoquée par le système lorsqu'un événement survient. Le module Python est intégré dans TigerJython mais peut également être téléchargé **depuis ce lien** pour être étudié ou utilisé en dehors de TigerJython.

Le serveur est démarré par la création d'un objet *TCPServer* qui spécifie le port IP sur lequel écouter ainsi que la fonction de rappel *onStateChanged()* à utiliser pour gérer les événements. Il passe ensuite en mode écoute.

```
from tcpcom import TCPServer
server = TCPServer(port, stateChanged = onStateChanged)
```

La fonction de rappel *onStateChanged (state, msg)* prend deux chaines de caractères en paramètre : *state* et *msg* qui décrivent le changement d'état du serveur:

state	msg	Description
TCPServer.LISTENING	port	Une connexion vient d'être terminée ou le serveur vient d'être démarré. Le serveur est en écoute d'une nouvelle connexion.

Server.PORT_IN_USE	port	Le serveur ne peut pas passer en mode écoute car le port est déjà occupé par un autre processus.
TCPServer.CONNECTED	Adresse IP du client	Un client a initié une connexion qui été acceptée
TCPServer.MESSAGE	Message reçu	Le serveur a reçu un message sur le socket
TCPSever.TERMINATED	(Vide)	Le serveur a terminé son exécution et n'est plus en écoute

Le client démarre avec la création d'un objet *TCPClient* spécifiant l'adresse et le port IP du serveur ainsi que la fonction de rappel *onStateChanged ()*. En invoquant *connect()*, le client démarre un tentative de connexion.

```
from tcpcom import TCPClient
client = TCPClient(host, port, stateChanged = onStateChanged)
client.connect()
```

Là encore, la fonction de rappel *onStateChanged (state, msg)* prend deux chaines de caractères *state* et *msg* en paramètre qui décrivent le changement d'état du client :

state	**msg**	**Description**
TCPClient.CONNECTING	Adresse IP du serveur	Démarre une tentative de connexion
TCPClient.CONNECTION_FAILED	Adresse IP du serveur	La tentative de connexion a échoué
TCPClient.CONNECTED	Adresse IP du serveur	La connexion est établie
TCPClient.MESSAGE	Message reçu	Le client a reçu un message
TCPClient.DISCONNECTED	(Vide)	La connexion a été interrompue par le client ou le serveur

L'appel à la fonction *connect()* est bloquant, ce qui signifie qu'elle va retourner *True* une fois la connexion établie ou *False* après un temps d'attente d'environ 10 secondes si la connexion échoue. L'information de succès ou d'échec de la connexion peut également être détectée à l'aide de la fonction de rappel.

On peut tester le programme client / serveur suivant sur la même machine en démarrant deux fenêtre TigerJython différentes et en exécutant le programme dans chacune d'elles. Dans ce cas, on choisit *localhost* comme nom d'hôte à savoir l'adresse IP 127.0.0.1 correspondant à l'interface réseau locale. Il est bien entendu plus réaliste d'utiliser deux machines différentes. Elles doivent alors être connectées par un réseau filaire ou Wi-Fi et la communication TCP/IP doit être autorisée sur le port sélectionné (attention au pare-feu). Un échec de la connexion par l'intermédiaire d'une borne Wi-Fi est le plus souvent dû à des restrictions du pare-feu qui y est intégré. En cas de problème, il faut se connecter à l'interface d'administration de la borne si vous en avez la possibilité ou activer le partage de la connexion de votre smartphone, ce qui va transformer ce dernier en un point d'accès mobile qui agira comme une borne Wi-Fi sans protection particulière. Il n'est pas nécessaire pour cela d'être connecté à Internet par le réseau mobile 3G ou similaire.

Votre première tâche de programmation réseau est de mettre en place un service réseau qui indique l'heure. Lorsqu'un client se connecte, ce service retourne le temps et la date actuels au client. Il existe des tas de tels serveurs de temps sur Internet et vous pouvez être fiers d'être

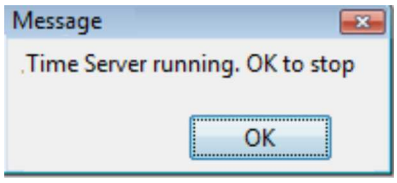
déjà en mesure de coder une application serveur professionnelle.

Pour éteindre le serveur de temps, on utilise une astuce bien connue qui consiste à « figer » le programme serveur dans une boîte de dialogue modale ouverte avec la fonction bloquante *msgDlg()*. Lorsque la fonction retourne lors d'un clic sur le bouton OK, le serveur est arrêté par un appel à *terminate()*.

```
from tcpcom import TCPServer
import datetime

def onStateChanged(state, msg):
    print state, msg
    if state == TCPServer.CONNECTED:
        server.sendMessage(str(datetime.datetime.now()))
        server.disconnect()

port = 5000
server = TCPServer(port, stateChanged = onStateChanged)
msgDlg("Time Server running. OK to stop")
server.terminate()
```

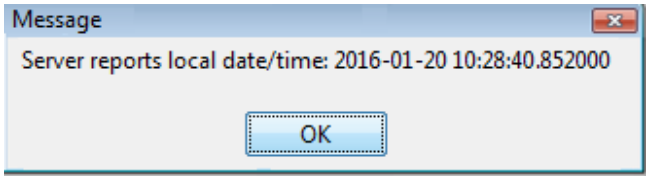
Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

Le client commence par demander l'adresse IP sur laquelle se connecter et effectue une tentative de connexion par un appel à *connect()*. L'information de temps reçue du serveur est écrite dans une boîte de dialogue.




```
from tcpcom import TCPClient

def onStateChanged(state, msg):
    print state, msg
    if state == TCPClient.MESSAGE:
        msgDlg("Server reports local date/time: " + msg)
    if state == TCPClient.CONNECTION_FAILED:
        msgDlg("Server " + host + " not available")

host = inputString("Time Server IP Address?")
port = 5000
client = TCPClient(host, port, stateChanged = onStateChanged)
client.connect()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)



MEMENTO

Le serveur et le client utilisent le modèle de la programmation événementielle avec la fonction de rappel *onStateChanged(state, msg)*. Les deux paramètres pris par *onStateChanged* contiennent des informations importantes concernant l'événement survenu. Il faut s'assurer de terminer l'exécution du serveur avec *terminate()* pour libérer le port IP utilisé.

SERVEUR ECHO

Vos prochains pas en programmation socket touchent à un scénario célèbre puisqu'il constitue l'archétype même de la communication client-serveur. Le serveur ne fait rien d'autre que de renvoyer au client les données qu'il a lui-même envoyées, sans les modifier. On appelle cela un serveur « écho ». Ce serveur peut ensuite facilement être modifié pour analyser les messages reçus du client et lui retourner des réponses en conséquence. C'est d'ailleurs exactement ce qui se passe avec tous les serveurs Web qui retournent une réponse à une requête HTTP initiée par le navigateur client.

On commence par coder le serveur pour le démarrer immédiatement. Le code est très similaire à celui du serveur de temps. Pour bien comprendre la communication et dans un but de débogage, le programme affiche dans la console les paramètres *state* et *msg* reçus par la fonction de rappel.

```
from tcpcom import TCPServer

def onStateChanged(state, msg):
    print state, msg
    if state == TCPServer.MESSAGE:
        server.sendMessage(msg)

port = 5000
server = TCPServer(port, stateChanged = onStateChanged)
msgDlg("Echo Server running. OK to stop")
server.terminate()
```

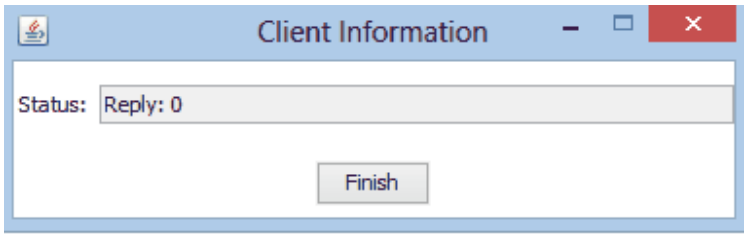
Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

Le client demande un peu plus d'efforts et utilise la classe *EntryDialog* pour afficher une boîte de dialogue non modale mettant en évidence les informations de changement d'état. La classe *EntryDialog* est également très appropriée dans d'autres situations puisqu'elle permet facilement d'ajouter des champs textuels éditables ou non éditables ainsi que d'autres composants graphiques tels que les boutons ou les curseurs.

```
from tcpcom import TCPClient
from entrydialog import *
import time

def onStateChanged(state, msg):
    print state, msg
    if state == TCPClient.MESSAGE:
        status.setValue("Reply: " + msg)
    if state == TCPClient.DISCONNECTED:
        status.setValue("Server died")

def showStatusDialog():
    global dlg, btn, status
    status = StringEntry("Status: ")
    status.setEditable(False)
    pane1 = EntryPane(status)
    btn = ButtonEntry("Finish")
    pane2 = EntryPane(btn)
    dlg = EntryDialog(pane1, pane2)
    dlg.setTitle("Client Information")
    dlg.show()

host = "localhost"
port = 5000
showStatusDialog()
client = TCPClient(host, port, stateChanged = onStateChanged)
status.setValue("Trying to connect to " + host + ":" + str(port) + "...")
time.sleep(2)
rc = client.connect()
if rc:
    time.sleep(2)
```

```
    n = 0
    while not dlg.isDisposed():
        if client.isConnected():
            status.setValue("Sending: " + str(n))
            time.sleep(0.5)
            client.sendMessage(str(n), 5)  # block for max 5 s
            n += 1
        if btn.isTouched():
            dlg.dispose()
        time.sleep(0.5)
    client.disconnect()
else:
    status.setValue("Connection failed.")
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)



MEMENTO

Dans le programme client, on utilise la fonction *sendMessage(msg, timeout)* avec un paramètre additionnel *timeout*. L'appel est bloquant pour un délai maximal spécifié en secondes en attendant que le serveur retourne une réponse. La fonction *sendMessage()* retourne la réponse du serveur ou *None* si aucune réponse n'est reçue dans le délai fixé.

Il est important de connaître la différence entre une boîte de dialogue modale et une boîte de dialogue non modale. Alors que la fenêtre modale bloque le programme jusqu'à ce qu'elle soit fermée, le programme poursuit son exécution sans problème avec une boîte de dialogue non modale, ce qui permet au programme d'afficher à tout moment des informations d'état ou de lire une saisie utilisateur.

JEU À DEUX JOUEURS EN LIGNE AVEC LES GRAPHIQUES TORTUE

Le « touché coulé » est un jeu populaire joué par deux joueurs dans lequel la mémorisation joue un rôle de premier plan. Le plateau de jeu de chacun des joueurs est un arrangement à une ou deux dimensions de cellules sur lesquelles sont disposés des vaisseaux qui occupent une ou plusieurs cases et qui possèdent une valeur différente selon le navire. À tour de rôle, chacun des joueurs désigne une case du jeu adverse sur laquelle une bombe va être larguée. Si la cellule visée contient une partie de navire, celle-ci va être touchée et supprimée du plateau de jeu (ou, dans certaines variantes, marquée comme touchée) et sa valeur sera créditée à l'attaquant. Si l'un des joueurs n'a plus de navire, la partie touche à sa fin et le joueur disposant du plus de points remporte la partie.

Dans sa version la plus simple, les navires sont affichés comme des carrés colorés dans un tableau unidimensionnel. Tous les navires sont de rang égal et le gagnant est le premier jour à éliminer tous les vaisseaux ennemis.

Du point de vue de la logique du jeu, le client et le serveur sont pratiquement identiques. Afficher le plateau de jeu ne nécessite rien d'autre que des primitives graphiques des tortues graphiques que vous maitrisez en principe depuis bien longtemps. On sélectionne d'abord aléatoirement un numéro de cellule entre 1 et 10 puis quatre nombres aléatoires différents désignant les cellules comportant un navire. La fonction *random.sample()* permet de faire ceci de manière élégante. Pour larguer une bombe, on envoie directement l'instruction Python appropriée sous forme de

chaine de caractères qui sera évaluée par le partenaire à l'aide de la fonction *exec()*. Une telle évaluation du code n'est possible que dans quelques rares langages de programmation dynamiques. [**plus...**] Pour savoir s'il y a eu un impact, il faut tester la couleur de fond avec *getPixelColorStr()*.

Dans le jeu, les deux joueurs sont sur un pied d'égalité mais leur programme diffère légèrement suivant qu'il joue le rôle du serveur ou du client. De plus, le serveur doit démarrer en premier.

```
from gturtle import *
from tcpcom import TCPServer
import random

def initGame():
    clear("white")
    for x in range(-250, 250, 50):
        setPos(x, 0)
        setFillColor("gray")
        startPath()
        repeat 4:
            forward(50)
            right(90)
        fillPath()

def createShips():
    setFillColor("red")
    li = random.sample(range(1, 10), 4) # 4 unique random numbers
    for i in li:
        fill(-275 + i * 50, 25)

def onMouseHit(x, y):
    global isMyTurn
    setPos(x, y)
    if getPixelColorStr() == "white" or isOver or not isMyTurn:
        return
    server.sendMessage("setPos(" + str(x) + "," + str(y) + ")")
    isMyTurn = False

def onCloseClicked():
    server.terminate()
    dispose()

def onStateChanged(state, msg):
    global isMyTurn, myHits, partnerHits
    if state == TCPServer.LISTENING:
        setStatusText("Waiting for game partner...")
        initGame()
    if state == TCPServer.CONNECTED:
        setStatusText("Partner entered my game room")
        createShips()
    if state == TCPServer.MESSAGE:
        if msg == "hit":
            myHits += 1
            setStatusText("Hit! Partner's remaining fleet size "
                + str(4 - myHits))
            if myHits == 4:
                setStatusText("Game over, You won!")
                isOver = True
        elif msg == "miss":
            setStatusText("Miss! Partner's remaining fleet size "
                + str(4 - myHits))
        else:
            exec(msg)
            if getPixelColorStr() != "gray":
                server.sendMessage("hit")
                setFillColor("gray")
                fill()
                partnerHits += 1
                if partnerHits == 4:
```

```
                    setStatusText("Game over, Play partner won!")
                    isOver = True
                    return
            else:
                server.sendMessage("miss")
            setStatusText("Make your move")
            isMyTurn = True

makeTurtle(mouseHit = onMouseHit, closeClicked = onCloseClicked)
addStatusBar(30)
hideTurtle()
port = 5000
server = TCPServer(port, stateChanged = onStateChanged)
isOver = False
isMyTurn = False
myHits = 0
partnerHits = 0
```

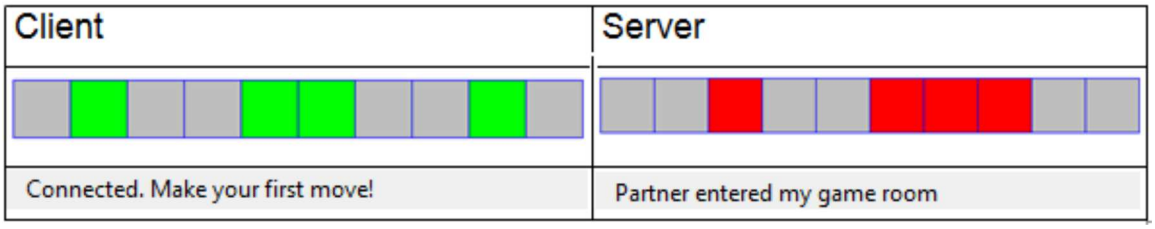
Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

The client program is almost the same:

```
from gturtle import *
import random
from tcpcom import TCPClient

def initGame():
    for x in range(-250, 250, 50):
        setPos(x, 0)
        setFillColor("gray")
        startPath()
        repeat 4:
            forward(50)
            right(90)
        fillPath()

def createShips():
    setFillColor("green")
    li = random.sample(range(1, 10), 4) # 4 unique random numbers 1..10
    for i in li:
        fill(-275 + i * 50, 25)

def onMouseHit(x, y):
    global isMyTurn
    setPos(x, y)
    if getPixelColorStr() == "white" or isOver or not isMyTurn:
        return
    client.sendMessage("setPos(" + str(x) + "," + str(y) + ")")
    isMyTurn = False

def onCloseClicked():
    client.disconnect()
    dispose()

def onStateChanged(state, msg):
    global isMyTurn, myHits, partnerHits
    if state == TCPClient.DISCONNECTED:
        setStatusText("Partner disappeared")
        initGame()
    elif state == TCPClient.MESSAGE:
        if msg == "hit":
            myHits += 1
            setStatusText("Hit! Partner's remaining fleet size "
                + str(4 - myHits))
            if myHits == 4:
                setStatusText("Game over, You won!")
                isOver = True
        elif msg == "miss":
            setStatusText("Miss! Partner's remaining fleet size "
```

```
                + str(4 - myHits))
        else:
            exec(msg)
            if getPixelColorStr() != "gray":
                client.sendMessage("hit")
                setFillColor("gray")
                fill()
                partnerHits += 1
                if partnerHits == 4:
                    setStatusText("Game over, Play partner won")
                    isOver = True
                    return
            else:
                client.sendMessage("miss")
            setStatusText("Make your move")
            isMyTurn = True

makeTurtle(mouseHit = onMouseHit, closeClicked = onCloseClicked)
addStatusBar(30)
hideTurtle()
initGame()
host = "localhost"
port = 5000
client = TCPClient(host, port, stateChanged = onStateChanged)
setStatusText("Client connecting...")
isOver = False
myHits = 0
partnerHits = 0
if client.connect():
    setStatusText("Connected. Make your first move!")
    createShips()
    isMyTurn = True
else:
    setStatusText("Server game room closed")
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)



MEMENTO

Dans le cas des jeux à deux joueurs, les joueurs peuvent partager un plateau commun ou, au contraire, avoir chacun une vision du jeu différente. Dans le second scénario qui inclut la plupart des jeux de cartes, le jeu doit nécessairement se jouer sur deux machines différentes puisqu'il faut garder son jeu secret. Au lieu de toujours laisser le client commencer la partie, le premier joueur pourrait être choisi au hasard ou par une négociation.

Les deux programmes se terminent par un clic sur le bouton « fermer » de la barre de titre. Il faut cependant encore faire explicitement le ménage pour terminer le serveur ou fermer le lien de communication. Pour ce faire, il faut enregistrer la fonction de rappel *onCloseClicked()* pour inhiber le comportement par défaut et implémenter un comportement personnalisé qui fait le ménage et ferme la fenêtre tortue avec un appel à *dispose()*. Le programmeur sans scrupule qui aura omis de faire le ménage correctement devra recourir au gestionnaire de tâches du système d'exploitation pour faire le ménage et pouvoir réutiliser le jeu sans encombre.

JEU EN LIGNE À DEUX JOUEURS AVEC GAMEGRID

Pour coder des jeux plus complexes, il est avantageux d'utiliser une bibliothèque de jeu plus perfectionnée qui simplifie considérablement le code. Nous avons déjà vu dans le **chapitre 7** comment utiliser *GameGrid*, un moteur de jeux complet intégré à TigerJython. En combinant la bibliothèque *GameGrid* avec *tcpcom*, il est possible de créer des jeux en ligne multijoueurs sophistiqués dans lesquels les partenaires de jeux sont situés aux extrémités du monde. En guise d'illustration, nous allons étendre le touché-coulé en deux dimensions. La logique du jeu demeure inchangée mais on transfère cette fois-ci les coordonnées X et Y de la cellule sélectionnée. Du côté de la réception, le partenaire peut déterminer si l'un de ses bateaux a été touché et retourner une réponse “hit” ou “miss”.

Dans le développement de jeux, il est important de consacrer un effort spécial pour faire en sorte que le jeu se comporte de manière raisonnable même si les deux joueurs ont un comportement quelque peu déraisonnable. Il faut par exemple interdire de larguer des bombes si ce n'est pas le tour du joueur. D'autre part, si l'un des joueurs quitte le jeu de manière inattendue, son partenaire devrait en être informé. Bien que ces précautions accroissent significativement le nombre de lignes de code, c’est là une marque distinctive d’une programmation scrupuleuse.

Puisqu'une grande partie du code est identique côté serveur et côté client et comme la duplication de code est l'un des plus grands péchés en programmation, le code commun est exporté dans un module *shiplib.py* qui peut ensuite être importé par les deux programmes. Les différences de comportement sont prises en compte par des paramètres supplémentaires tels que *node* qui fait référence à un objet *TCPServer* ou *TCPClient*.

Module séparé:

```
# shiplib.py

from gamegrid import *

isOver = False
isMyMove = False
dropLoc = None
myHits = 0
partnerHits = 0
nbShips = 2

class Ship(Actor):
     def __init__(self):
         Actor.__init__(self, "sprites/boat.gif")

def handleMousePress(node, loc):
    global isMyMove, dropLoc
    dropLoc = loc
    if not isMyMove or isOver:
          return
    node.sendMessage("" + str(dropLoc.x) + str(dropLoc.y)) # send location
    setStatusText("Bomb fired. Wait for result...")
    isMyMove = False

def handleMessage(node, state, msg):
    global isMyMove, myHits, partnerHits, first, isOver
    if msg == "hit":
        myHits += 1
        setStatusText("Hit! Partner's fleet size " + str(nbShips - myHits)
            + ". Wait for partner's move!")
        addActor(Actor("sprites/checkgreen.gif"), dropLoc)
        if myHits == nbShips:
            setStatusText("Game over, You won!")
            isOver = True
    elif msg == "miss":
        setStatusText("Miss! Partner's fleet size " + str(nbShips - myHits)
            + ". Wait for partner's move!")
        addActor(Actor("sprites/checkred.gif"), dropLoc)
    else:
```

```
        x = int(msg[0])
        y = int(msg[1])
        loc = Location(x, y)
        bomb = Actor("sprites/explosion.gif")
        addActor(bomb, loc)
        delay(2000)
        bomb.removeSelf()
        refresh()
        actor = getOneActorAt(loc, Ship)
        if actor != None:
            actor.removeSelf()
            refresh()
            node.sendMessage("hit")
            partnerHits += 1
            if partnerHits == nbShips:
                setStatusText("Game over! Partner won")
                isOver = True
                return
        else:
            node.sendMessage("miss")
        isMyMove = True
        setStatusText("You fire!")
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

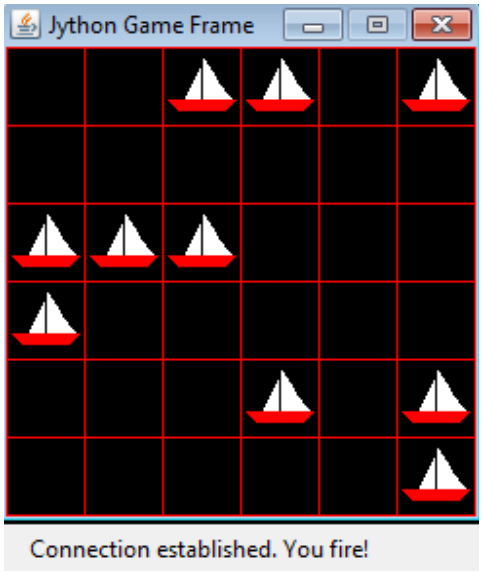
L'utilisation du module externe *shiplib.py* simplifie grandement le code du serveur et du client. Voici le code serveur :

```
from gamegrid import *
from tcpcom import TCPServer
import shiplib

def onMousePressed(e):
     loc = toLocationInGrid(e.getX(), e.getY())
     shiplib.handleMousePress(server, loc)

def onStateChanged(state, msg):
    global first
    if state == TCPServer.PORT_IN_USE:
        setStatusText("TCP port occupied. Restart IDE.")
    elif state == TCPServer.LISTENING:
        setStatusText("Waiting for a partner to play")
        if first:
            first = False
        else:
            removeAllActors()
            for i in range(shiplib.nbShips):
                addActor(shiplib.Ship(), getRandomEmptyLocation())
    elif state == TCPServer.CONNECTED:
        setStatusText("Client connected. Wait for partner's move!")
    elif state == TCPServer.MESSAGE:
        shiplib.handleMessage(server, state, msg)

def onNotifyExit():
    server.terminate()
    dispose()

makeGameGrid(6, 6, 50, Color.red, False, mousePressed = onMousePressed,
             notifyExit = onNotifyExit)
addStatusBar(30)
for i in range(shiplib.nbShips):
    addActor(shiplib.Ship(), getRandomEmptyLocation())
show()
port = 5000
first = True
server = TCPServer(port, stateChanged = onStateChanged)
shiplib.node = server
```

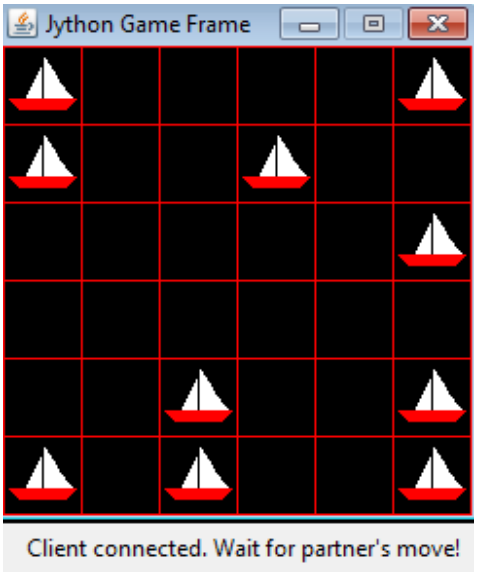
```
from gamegrid import *
from tcpcom import TCPClient
import shiplib

def onMousePressed(e):
     loc = toLocationInGrid(e.getX(), e.getY())
     shiplib.handleMousePress(client, loc)

def onStateChanged(state, msg):
    if state == TCPClient.CONNECTED:
        setStatusText("Connection established. You fire!")
        shiplib.isMyMove = True
    elif state == TCPClient.CONNECTION_FAILED:
        setStatusText("Connection failed")
    elif state == TCPClient.DISCONNECTED:
        setStatusText("Server died")
        shiplib.isMyMove = False
    elif state == TCPClient.MESSAGE:
        shiplib.handleMessage(client, state, msg)

def onNotifyExit():
    client.disconnect()
    dispose()

makeGameGrid(6, 6, 50, Color.red, False,
    mousePressed = onMousePressed, notifyExit = onNotifyExit)
addStatusBar(30)
for i in range(shiplib.nbShips):
    addActor(shiplib.Ship(), getRandomEmptyLocation())
show()
host = "localhost"
port = 5000
client = TCPClient(host, port, stateChanged = onStateChanged)
client.connect()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

client



serveur



MEMENTO

La condition de fin de jeu est une situation spéciale qui demande toujours au programmeur une pensée scrupuleuse. Puisqu'il s'agit d'une situation d'urgence, on utilise un fanion global *isOver* qui est mis à *True* lorsque le jeu est terminé. Il faut également se poser la question de savoir si une nouvelle partie doit pouvoir être démarrée sans qu'il faille redémarrer le programme serveur et le programme client. Dans l'implémentation présentée ci-dessous, le client doit être fermé et le serveur retourne en état d'attente de connexion pour démarrer une nouvelle partie. Les programmes pourraient être améliorés de plusieurs façons : de manière générale, la programmation est un domaine très attractif puisqu'il n'y a aucune limite à l'imagination et à l'ingéniosité des développeurs. De plus, après tout, se relaxer et jouer au jeu développé avec tant de peine est également très amusant.

Jusqu'à présent, les deux partenaires de jeu doivent tous deux démarrer leur propre programme selon la règle suivante: "le serveur en premier et ensuite le client". Pour contourner cette restriction, on peut user de l'astuce suivante: un programme démarre toujours en tant que client et essaie de créer une connexion au serveur. S'il échoue, il démarre en tant que serveur [**plus...**].

COMMUNICATION AVEC POIGNÉE DE MAINS

Même dans la vie de tous les jours, la communication entre deux partenaires nécessite une certaine forme de synchronisation. En particulier, l'envoi de données ne peut se faire que si le destinataire est prêt à la réception et au traitement des données. Ne pas observer cette règle pourrait conduire à une perte de données ou même au blocage des programmes. Il faut également prendre en compte la différence de puissance de calcul entre les deux nœuds ainsi qu'un délai de transmission changeant.

Une technique bien connue pour surmonter ces difficultés consiste, pour le destinataire, à retourner à l'émetteur une confirmation de réception, ce qui peut être comparé à une amicale poignée de mains. Le processus est relativement simple : des données sont transmises en bloc et le destinataire confirme qu'il les a bien reçues et qu'il est prêt pour la réception du prochain bloc. L'émetteur renverra alors le prochain bloc uniquement lorsqu'il aura reçu la confirmation de réception. Ce mécanisme de confirmation peut également conduire l'émetteur à renvoyer le même bloc de données une seconde fois [**plus...**].

Pour illustrer ce principe de la poignée de mains, le programme *turtle* serveur dessine assez lentement des lignes dictées par les clics de souris du client. Les mouvements de la tortue du client sont bien plus rapides. Le client doit de ce fait attendre entre chaque clic que le serveur confirme l'achèvement de son opération de dessin et qu'il soit prêt à accepter les prochaines commandes.

Serveur :

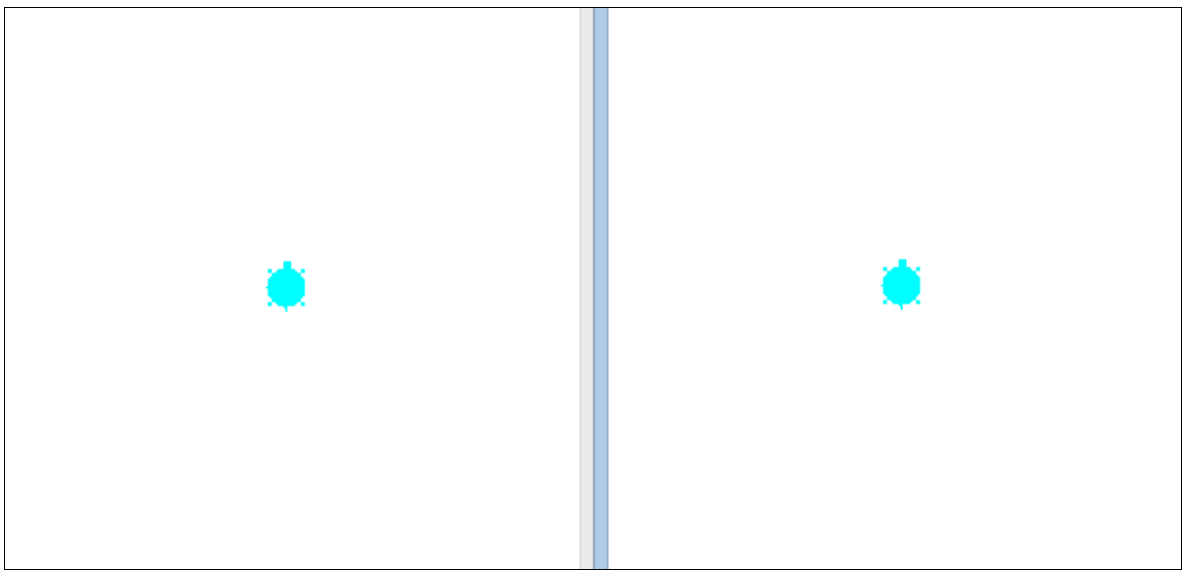
```
from gturtle import *
from tcpcom import TCPServer

def onCloseClicked():
    server.terminate()
    dispose()

def onStateChanged(state, msg):
    if state == TCPServer.MESSAGE:
        li = msg.split(",")
        x = float(li[0])
        y = float(li[1])
        moveTo(x, y)
        dot(10)
        server.sendMessage("ok")
```

```
makeTurtle(mouseHit = onMouseHit, closeClicked = onCloseClicked)
port = 5000
server = TCPServer(port, stateChanged = onStateChanged)
```

Client:

```
from gturtle import *
from tcpcom import TCPClient

def onMouseHit(x, y):
    global isReady
    if not isReady:
        return
    isReady = False
    client.sendMessage(str(x) + "," + str(y))
    moveTo(x, y)
    dot(10)

def onCloseClicked():
    client.disconnect()
    dispose()

def onStateChanged(state, msg):
   global isReady
   if state == TCPClient.MESSAGE:
        isReady = True

makeTurtle(mouseHit = onMouseHit, closeClicked = onCloseClicked)

speed(-1)
host = "localhost"
port = 5000
isReady = True
client = TCPClient(host, port, stateChanged = onStateChanged)
client.connect()
```

Server: Client:



MEMENTO

Afin de garantir que les actions de l'émetteur et du récepteur se déroulent dans le bon ordre, l'émetteur n'envoie le prochain paquet de données que lorsque le récepteur lui a confirmé qu'il est prêt à traiter les données suivantes..

EXERCICES

1a. Développer un système client-serveur dans lequel le serveur dispose de bonnes connaissances mathématiques. Le client doit pouvoir saisir dans une boîte de dialogue une expression impliquant des fonctions du module *math*, par exemple *sqrt(2)*, qui sera évaluée par le serveur. Le résultat de l'évaluation sera affiché dans la console du client. Pour réaliser l'évaluation du côté serveur, utilisez la fonction *exec()* intégrée à Python.

1b. Améliorer le programme précédent pour que le serveur retourne le message « illegal » s'il n'est pas capable d'évaluer l'expression demandée. Indication : utiliser une structure *try … except* pour attraper l'exception générée lorsque *exec()* n'est pas capable d'exécuter l'instruction demandée.

2. Développer un système client-serveur dans lequel le client peut envoyer des ordres que le serveur exécute. Le serveur démarre une fenêtre tortue et attend les ordres du client. Du côté client, l'utilisateur peut saisir une commande tortue qui sera envoyée au serveur pour exécution. Le serveur retourne le message « OK » en cas de succès et « fail » dans le cas contraire. Ce message sera affiché dans la console du client qui enverra l'instruction suivante uniquement sur réception de ce message de confirmation. Remarque : le client peut également envoyer plusieurs instructions séparées par un point-virgule en une seule fois.

3. Ajouter des effets sonores au touché-coulé à deux dimensions. Suggestion : émettre un son lorsque la bombe et larguée et lorsque la cible est manquée ou touchée. Utiliser les compétences vues dans le **chaptre 4**: son. Vous pouvez utiliser des sons prédéfinis, mais vos propres sons seront plus amusants.

4*. Améliorer le touché-coulé à une dimension de sorte que les bateaux ne soient plus représentés par des carrés colorés mais par des images qui disparaissent lorsqu'elles sont touchées.

MATÉRIEL SUPPLÉMENTAIRE

MESURES À DISTANCE AVEC LE RASPBERRY PI

Les communications client-serveur par TCP/IP jouent également un rôle très important dans la technologie de mesure et de contrôle. Il arrive bien souvent qu'un instruments de mesure ou qu'un robot se trouve éloigné de son centre de contrôle et que les données doivent donc être transmises par TCP/IP. Dans l'exercice 2, nous avons déjà réalisé un contrôle à distance. Dans l'exemple qui suit, nous utiliserons un Raspberry Pi agissant comme une interface entre le capteur de mesures et l'Internet en envoyant les données mesurées vers une station de collecte. Pour simplifier le système, nous n'utiliserons pas de réel capteur mais uniquement un bouton-poussoir dont l'état enfoncé ou relâché sera reporté.

Le programme de détection de mesures tournant sur le Raspberry Pi utilise le module *tcpcom* qu'il faut copier dans le même dossier que le programme sur le RPi. Pour tester le programme, on pourra utiliser un clavier et un écran directement rattachés au RPi ou piloter un PC distant à l'aide de SSH, WinSCP ou VNC. Pour plus d'informations, consulter la littérature ou faire une recherche sur Internet.

Le programme serveur est vraiment très simple : dans la boucle de mesure, on appelle périodiquement *GPIO.input()* pour interroger l'état du bouton-poussoir. La valeur 0 est retournée lorsque le bouton est enfoncé. L'information de l'état du bouton est ensuite transférée au client. Si le bouton est enfoncé durant trois cycles de mesure, le programme serveur est stoppé.

```
import time
import RPi.GPIO as GPIO
from tcpcom import TCPServer

def onStateChanged(state, msg):
    print "State:", state, "Msg:", msg

P_BUTTON1 = 16 # Switch pin number
dt = 1  # 1 s period
port = 5000 # IP port

GPIO.setmode(GPIO.BOARD)
GPIO.setwarnings(False)
GPIO.setup(P_BUTTON1, GPIO.IN, GPIO.PUD_UP)
server = TCPServer(port, stateChanged = onStateChanged)
n = 0
while True:
    if server.isConnected():
        rc = GPIO.input(P_BUTTON1)
        if rc == 0:
            server.sendMessage("pressed")
            n += 1
            if n == 3:
                break
        else:
            server.sendMessage("released")
            n = 0
        time.sleep(dt)
server.terminate()
print "Server terminated"
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

Le client se contente d'afficher les données reçues sur la sortie standard.

```
from tcpcom import TCPClient

def onStateChanged(state, msg):
    print "State: " + state + ". Message: " + msg

host = "192.168.0.5"
#host = inputString("Host Address?")
port = 5000 # IP port
client = TCPClient(host, port, stateChanged = onStateChanged)
rc = client.connect()
if rc:
    msgDlg("Connected. OK to terminate")
    client.disconnect()
```

Sélectionner le code (Ctrl+C pour copier, Ctrl+V pour coller)

CONTACT

Pour toute demande d'aide ou suggestion, écrire un courriel à l'adresse

help@tigerjython.com

Équipe de développement: Jarka Arnold, Haute École Pédagogique Berne
www.java-online.ch

Tobias Kohn, Gymnase de l'Oberland Zürichois
www.tobiaskohn.ch

Dr. Aegidius Plüss, Université de Berne
www.aplu.ch

Traduction française: Cédric Donner, Collège du Sud, Bulle

À propos des auteurs

Jarka Arnold
Jarka Arnold possède une longue expérience d'enseignement de l'informatique à la HEP Berne. Elle a supervisé le développement de plusieurs environnements d'apprentissage de la programmation dans le contexte de divers projets de recherche. Ces environnements sont utilisés avec succès par de nombreuses institutions éducatives (**http://www.java-online.ch** et **http://www.jython.ch**).

Tobias Kohn
Tobias Kohn (**http://www.tobiaskohn.ch/**) a terminé ses études de mathématiques à l'EPFZ (ETHZ) en 2008 et travaille depuis lors comme enseignant de mathématiques et d'informatique au Gymnase de l'Oberland zürichois à Wetzikon. En 2012, il débute une thèse doctorale à l'ETHZ en plus de son enseignement et cherche des moyens pour simplifier les cours d'introduction à la programmation.

Aegidius Plüss
Aegidius Plüss (**http://www.aplu.ch**) est un professeur émérite d'informatique et de didactique de l'informatique à l'Université de Berne. Il a écrit l'ouvrage "Java exemplarisch" et s'est impliqué dans plusieurs cours de formation continue destinés aux enseignants d'informatique suisses. Il développe des bibliothèques très fournies ainsi que des environnements de programmation pour les cours d'informatique.

Cédric Donner
Cédric Donner (**https://www.linkedin.com/in/cedric-donner**) a terminé ses études de mathématiques / physique à l'Université de Fribourg en 2008. Il enseigne depuis lors les mathématiques et l'informatique au Collège du Sud (Gymnase) à Bulle, FR. Il s'intéresse particulièrement aux techniques modernes de développement Web dans le but de créer des moyens d'enseignement en ligne pour l'informatique et les mathématiques au gymnase.