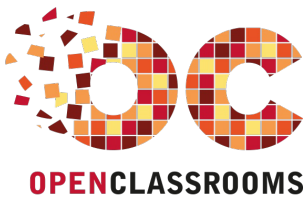
Langage Python

Par Suzy



www.openclassrooms.com

Dernière mise à jour le 24/08/2011

Sommaire

Langage Python

Par  Suzy

Mise à jour : 24/08/2011
Difficulté : Facile

Bienvenue sur le tutoriel Python !

Le Python est un animal qui peut s'avérer dangereux mais aussi le nom d'un langage de programmation très puissant qui est de plus en plus utilisé pour développer des applications rapidement qui soient tout de même efficaces ! Il est très utilisé dans le monde scientifique pour sa syntaxe aérée et une vitesse d'exécution correcte. Ce langage est aussi excellent pour la création de prototypes car sa simplicité permet d'implémenter un projet aussi vite que l'on y réfléchit ! Il est aussi utilisé en modélisation 3D dans des programmes comme Blender par exemple. Bref, c'est un langage de plus en plus utilisé et vous en deviendrez très certainement fan rapidement !

Les principales caractéristiques du langage Python sont :

- C'est un **langage interprété**, c'est-à-dire que le code ne nécessite pas d'être compilé pour être exécuté (nous verrons ça plus tard, ne vous en faites pas !)
- Il est **multiplateforme** et il est d'ailleurs natif sur tous les systèmes Unix (Mac OS, Linux) et ne nécessite pas d'y être installé !
- Il peut être utilisé en tant que **langage de script** pour exécuter une suite simple de commande mais c'est aussi un **langage objet** qui permet de développer des applications solidement construites !
- Sa syntaxe est très simple et aérée et permet de se libérer de tous les caractères de démarquation des blocs de code.

Hein ?! Qu'est-ce qu'il dit le monsieur ?!

Vous n'avez pas tout compris aux caractéristiques du langage Python écrites ci-dessus ? Ce n'est pas un problème, ce cours est là pour tout vous expliquer de zéro ! Allez, c'est parti !

Il existe différentes versions de Python avec notamment quelques différences syntaxiques en Python 2 et Python 3. J'essayerai au maximum de présenter les scripts dans les deux versions afin que chacun puisse y trouver son compte. Faites donc bien attention de regarder dans quelle version chaque portion de code est écrite.

N'hésitez surtout pas à descendre dans les tréfonds du tutoriel pour y trouver des exercices et des annexes !

Partie 1 : Bases et programmation impérative

Alors, nous allons y aller doucement pour ne pas rebuter les néophytes. Dans cette partie, vous apprendrez à commencer à programmer en Python avec tout d'abord son installation, l'utilisation du shell et l'écriture des premiers scripts ! Vous ne ferez que de la programmation impérative, c'est-à-dire que vous n'écrirez que des suites d'instructions s'exécutant les unes à la suite des autres.

Bien commencer

Vous êtes venu ici pour apprendre à programmer mais seulement, savez-vous ce que cela veut dire ? Et savez-vous à quoi cela sert ? C'est ce que nous allons voir, alors au travail ! Mais avant de pouvoir se mettre réellement au travail, il va d'abord falloir installer ce qu'il faut pour travailler ! Et pour pouvoir programmer en Python, un éditeur de texte suffit. Seulement, pour exécuter du code, vous aurez besoin d'un interpréteur.

Qu'est-ce qu'un interpréteur ? Et à quoi ça sert ?

L'interpréteur est le programme qui va "interpréter" votre code, c'est à dire qu'il va traduire ce que vous avez écrit en code machine, c'est-à-dire une suite de 0 et de 1 que votre ordinateur peut comprendre. C'est lui qui va vous permettre de lancer vos programmes.
Dans cette section, nous allons voir :

- A quoi ça sert de programmer ?
- Comment installer Python
- Comment utiliser le shell Python

Les programmeurs en herbe qui travaillent sous Mac OS ou Linux seront ravis d'apprendre que, pour la plupart, ils n'auront pas besoin d'installer Python sur leur ordinateur car les programmes développés en Python pourront y être exécuté sans installer quoi que ce soit ! Mais nous verrons tout de même comment accéder au shell et surtout à quoi ça sert ! Commençons avant que vous demandiez ce qu'est un shell... Nous le verrons d'ici la fin du chapitre.

Programmer

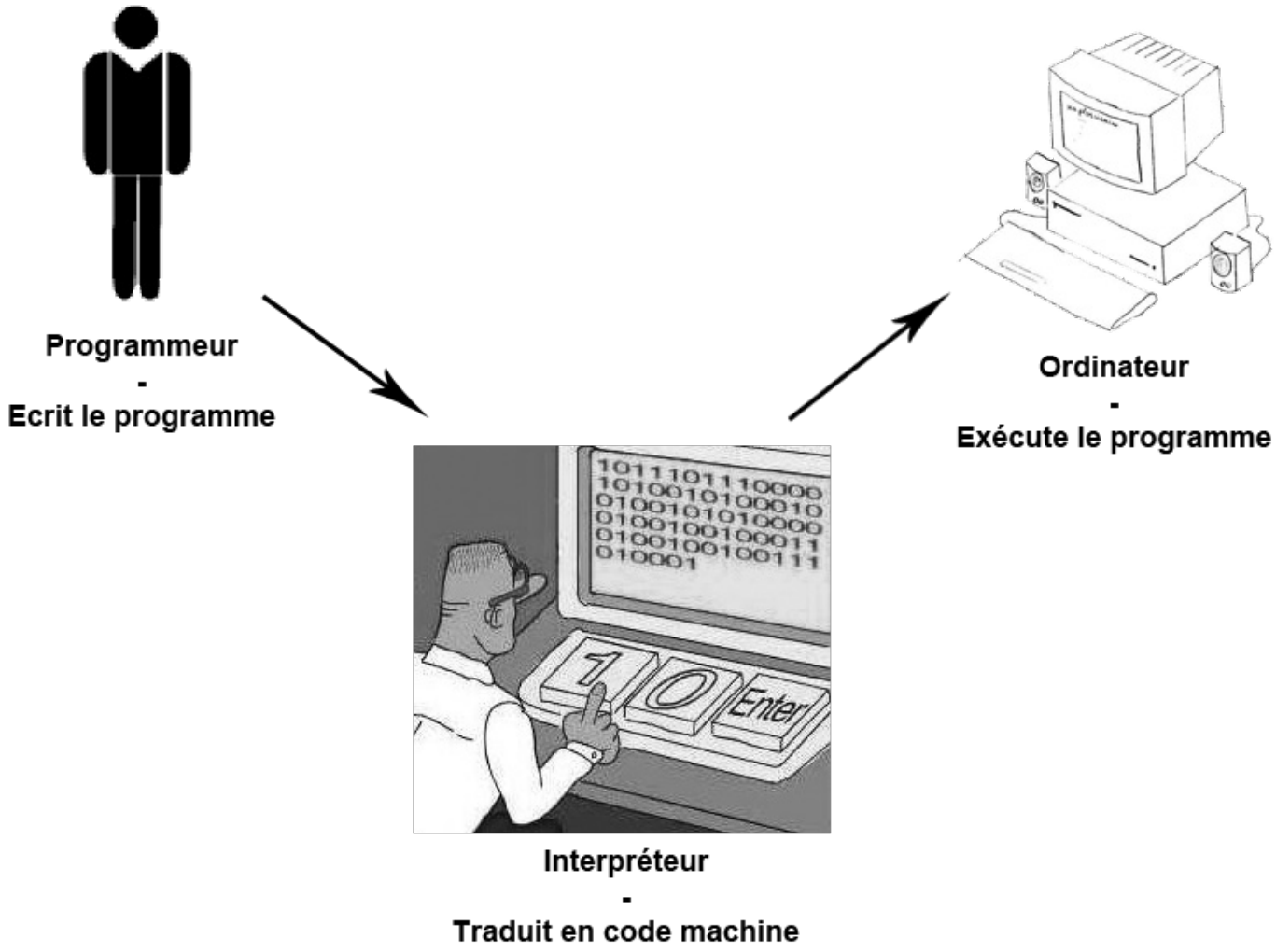
Lorsque vous utilisez un ordinateur, vous pouvez faire plein de choses ! Chacune des actions que vous effectuez avec votre ordinateur a été programmée ! En effet, tout ce qui rend une machine utilisable est un programme, une suite d'instructions qui ont été écrites par quelqu'un (ou quelque chose, mais ce n'est pas encore de votre niveau...) et qui s'exécutent. Les ordinateurs, et plus particulièrement les processeurs qui sont en quelque sorte le cerveau, ne sont capables de comprendre qu'une suite de 0 et 1 qui n'ont pas de sens pour un humain normal...

Ça veut dire que tout ce que je fais sur mon ordinateur n'est qu'une suite de 0 et de 1 ?!
Mais alors, comment ça se fait que je peux lire quelque chose sur mon ordinateur, écouter de la musique, et jouer à des jeux débiles sur internet ?!

Et bien, c'est là tout le sens du mot informatique : c'est la science de l'information. Tous les composants de votre ordinateur ont été créés pour être capables de reconnaître des suites de 0 et 1 et de les transformer en quelque chose de compréhensible pour vous. Une suite de 0 et 1 sera interprétée par votre écran comme un ensemble de couleurs à faire afficher à chacun des pixels qui le composent par exemple. Votre carte son transformera ça en signal électrique qui ira faire vibrer vos enceintes pour que vous puissiez écouter votre musique préférée.

Il y a des gens qui écrivent des 0 et des 1 toute la journée que l'on puisse écouter de la musique et faire n'importe quoi avec notre ordinateur ?!

Cela a existé à une époque, au début de l'informatique, lorsque l'on faisait une calculatrice pour collège avec un ordinateur de la taille d'un immeuble... Mais aujourd'hui, les programmes que l'on utilise sont très complexes et permettent de faire bien des choses ! Et c'est là que la programmation, celle avec un langage de programmation intervient ! Un langage de programmation, c'est en fait un langage spécifique, assez simple et intelligible par des humains composé d'une syntaxe particulière (ensemble de mots clés), qui sera ensuite traduit en une suite gigantesque de 0 et de 1 pour que la machine puisse à son tour le comprendre et surtout l'exécuter. C'est donc une manière beaucoup plus simple pour nous de faire faire quelque chose à un ordinateur. L'inconvénient est qu'il n'est pas possible d'optimiser au maximum les opérations que nous allons lui demander de faire. Il y a toujours une déformation due à la traduction. Les machines se faisant de plus en plus puissantes, cela ne pose plus de gros problèmes aujourd'hui. En résumé, ce que vous faites lorsque vous programmez peut ressembler à ça :



Des centaines de langages si ce n'est des milliers ont été créés depuis les débuts de l'informatique. Le site 99 bottles of beer en répertorie un assez grand nombre. Python est loin d'être le seul langage mais dans ce cours, nous ne verrons que celui-ci ! Maintenant que vous savez à quoi cela peut bien servir de programmer, vous aimeriez sans doute savoir comment il est possible de le faire ! Mais pas si vite, comme tout travailleur qui se respecte, il faut avoir les bons outils ! Et cela commence par le traducteur entre vous et la machine : **l'interpréteur**.

Installer Python

Sous Windows

Pour les utilisateurs de Windows (dont je fais partie), le langage Python ne peut pas être reconnu sans l'installation d'un programme tiers appelé **interpréteur**. Pour vous procurer un tel programme, il suffit d'aller sur le site officiel de Python. Pour les non anglophones (mais il va falloir vous y mettre !), il faut aller ensuite sur la page appelée "downloads" (téléchargements). De là, il suffit de télécharger la version qui vous convient (pour ma part, j'utiliserai la version 3.1) dans les fichiers proposés pour Windows.

Une fois le bon fichier téléchargé, il suffit de l'exécuter pour avoir l'installateur ! Vous devez d'abord choisir l'emplacement où l'installer sur votre ordinateur. Personnellement je laisse l'adresse par défaut, ça ne prend pas beaucoup d'espace et au moins c'est bien rangé. Vous aurez ensuite le choix d'installer ou non certaines parties de Python. Voici ce que vous devez avoir sous les yeux (hormis mon tutoriel) :

Ici, vous avez donc le choix d'installer Python mais aussi quelques modules complémentaires. Alors qu'allons nous retrouver une fois tout ça installé ?

- **Python** : bah oui, à la base, c'est ce que nous voulions installer...
- **Tcl/Tk** : c'est un module qui vous permettra de faire des interfaces graphiques pour vos programmes
- **La documentation** : nous verrons plus tard comment l'utiliser correctement et à quoi cela peut servir, mais sachez que cela pourra vous permettre de programmer sans connexion internet !
- **IDLE** : c'est un programme qui vous permettra d'éditer vos fichiers Python et de lancer vos programmes. C'est un peu basique mais largement suffisant pour un début !

Bon, maintenant cliquez sur "next" (suivant) et rien de plus compliqué ne devrait vous être demandé. Python est maintenant installé sur votre ordinateur !

Sous Mac OS ou Linux

L'interpréteur Python est fourni sur la plupart des distributions Mac OS et Linux. Il vous permettra donc d'exécuter les scripts Python sans rien y installer. Cependant, si vous voulez vérifier qu'il est bien présent sur votre machine, vous pouvez taper ceci en console :

Code : Console

```
python
```

Si cette commande n'est pas reconnue, je vous conseille vivement de faire un tour sur le site officiel pour télécharger la dernière version de Python. Pour les autres, vérifiez le numéro de version qu'il vous affiche. Si vous n'avez pas la version 2.7 ou la version 3.1 de Python, je vous conseille alors de mettre à jour votre version soit en allant sur la même page que pour les windowsiens (voir ci-dessus) si vous êtes sur Mac OS et sinon, il suffit de récupérer les paquets via internet si vous êtes sous Linux. Pour ceux qui travaillent sur Mac, le fait d'installer ou de mettre à jour votre version de Python vous permettra d'être sûr de posséder le petit environnement de développement fourni avec : IDLE. Il vous permettra d'écrire vos programmes mais aussi de les lancer pour les tester.

Sous autre chose

Python est réellement multiplateforme et en plus d'être disponible pour les systèmes sous Windows, Mac OS et Linux, il peut aussi être utilisé pour programmer sur votre PSP, vos iPod et bien d'autres ! Si cela vous intéresse, leur site propose les liens vers les sites officiels des versions existants pour divers OS à cette adresse : http://www.python.org/download/other/

Bien, maintenant que vous avez les outils de base pour travailler, nous allons voir comment utiliser Python.

Utiliser le shell

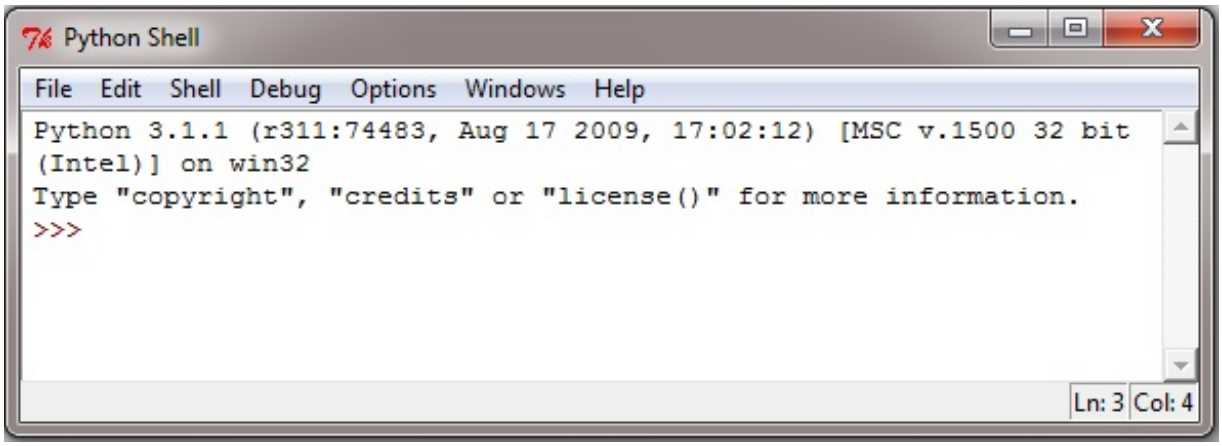
Python est un langage interprété, c'est-à-dire que vous n'avez pas besoin de traduire votre code en langage machine, vous avez un traducteur pour ça, l'interpréteur, qui transformera à la volée le code que vous donnerez à manger à votre ordinateur ! De ce fait, Python propose un outil sympathique appelé **shell**.

Hein quoi ? Un shell ? C'est quoi ? Ca se mange ?

Meuh non ! Un shell, c'est en fait une console spécifique dans laquelle vous pourrez taper directement du code Python qui sera immédiatement reconnu. Pour accéder au shell, il faut :

- **Sous Windows ou Mac OS** : lancer IDLE (installé avec Python)
- **Sous Mac OS ou Linux** : taper "python" dans une console

Vous devriez obtenir une console qui ressemble à cela :



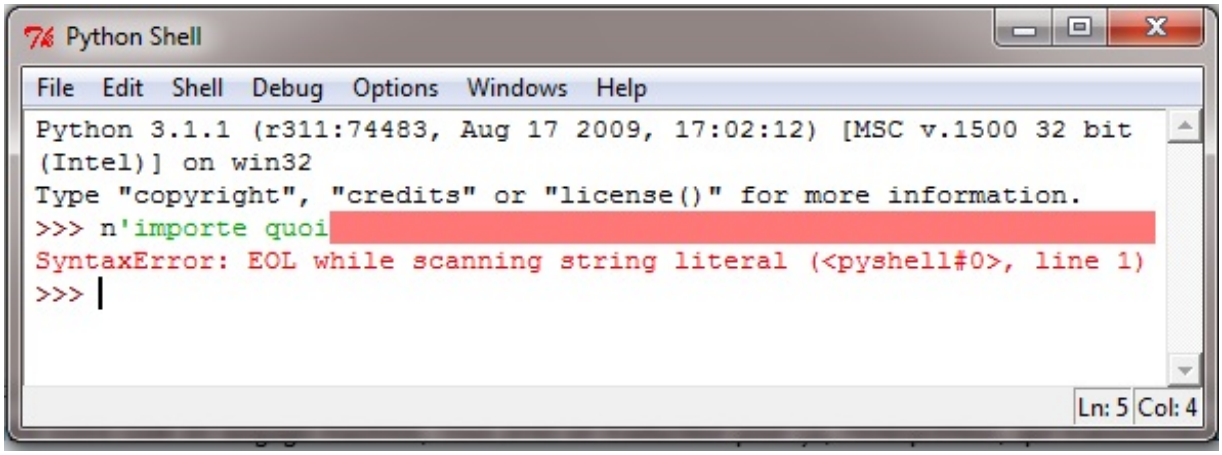
On peut voir ici que ma version de Python y est écrite. Elle m'indique donc que je suis sur un système sous WIndows 32bits avec une version 3.1.1 de Python.

Okay, j'ai le shell, mais j'en fais quoi moi de ça ?! Je vois pas en quoi je programme ici !

Eh bien maintenant que le shell est lancé, vous pouvez y taper tout ce que vous voulez du moment que c'est en python ! Nous allons faire quelques tests...

Taper n'importe quoi

Commençons par faire les animaux intrigués face à cet objet étrange qu'est le shell. Essayer de taper n'importe quoi. Vous devriez obtenir quelque chose dans le genre...



Comme je lui ai littéralement tapé "n'importe quoi", l'interpréteur me signale par une belle couleur rouge et un petit texte en anglais qu'il ne comprend pas ce que je lui demande de faire. C'est bon signe ! Ainsi, ici, il me dit qu'une ligne s'est terminée (*EOL : end of line*) prématurément alors qu'il essayait de comprendre (*while scanning*) une chaîne de caractères (*string*).

Wohohoh ! Tu veux dire que ce truc ne parle que anglais ?!

Et oui... Je vous avait dit plus haut qu'il allait falloir vous mettre à l'anglais... Tous les langages de programmation sont en fait une sorte d'anglais simplifié... Et tous les compilateurs et interpréteurs que vous pourrez utiliser vous écriront en anglais lorsqu'ils n'aimeront pas ce que vous leur demandez de traduire... Enfin, je vous rassure, des fois, même en parlant anglais, il arrive que ce qui dérange l'interpréteur soit complètement incompréhensible... Mais ça, vous apprendrez à reconnaître vos erreurs avec l'expérience ! Pour ce qui est de l'erreur, vous comprendrez plus tard...

Une grosse calculette

Maintenant que l'on s'est bien amusé à taper n'importe quoi dans notre shell, il va peut-être falloir se mettre au travail un peu plus sérieusement ! Pour ça, nous allons voir que Python peut servir de grosse calculatrice ! Essayer par exemple de lui donner à faire un gros calcul avec les quatre opérations de base (+, -, *, /) du style :

Code : Console - Shell Python

```
>>> 18*4*34*99 + 23455669992 - 23993002
23431919342
```

Je lui ai donné à faire ici un gros calcul et comme il est de bonne humeur, il m'affiche le résultat. Vous voyez donc que l'on peut travailler avec des nombres, c'est d'ailleurs la base de tout ce que l'on fera ! Les opérateurs que vous pouvez utiliser pour les nombres sont les suivants :

Symbole	Effet	Exemple
+	Addition	2 + 2 = 4
-	Soustraction	2 - 2 = 0
*	Multiplication	3 * 2 = 6
/	Division	5 / 2 = 2.5
**	Puissance	3 ** 3 = 27
//	Division entière (Euclidienne)	5 // 2 = 2
%	Modulo (reste de la division)	5 % 2 = 1

Dans certains langages et même suivant la version de Python (avant la version 3) que vous utiliserez, les opérations '/'

et '//' donneront le même résultat ! En effet, la conversion automatique en nombre à virgule lorsque cela est utile est très récent !

Si vous n'êtes pas familiers avec les opérations de division entière (ou dite Euclidienne) et le modulo, je vous conseille d'apprendre à vous en servir. Ce sont des opérations pratiques et qui, contrairement à ce que l'on peut penser initialement, sont utilisées régulièrement. Des exercices seront proposés pour mettre en évidence toute l'utilité de ces opérations dans la suite du cours....
Très bien, maintenant que Python est installé sur votre ordinateur et que vous avez eu un premier contact avec le shell qui vous servira pour tous vos petits tests rapides, on va pouvoir attaquer la suite ! Mais ne vous inquiétez pas, nous allons y aller en douceur.

Premier programme

Vous vous êtes bien amusé ? Nous allons attaquer notre premier programme ! Ca fait peur, hein ? Ne vous inquiétez surtout pas, nous allons commencer par quelque chose de très simple.

Pour rester dans le classique, on va se contenter de faire le programme Hello World. Pour ceux qui ne sont pas familiers avec la programmation, ce programme est le premier que tous ceux qui ont eu à apprendre un langage de programmation ont fait ! C'est un programme relativement simple qui consiste à écrire "Hello world !" dans la console...

Ce n'est donc pas encore ici que vous deviendrez des pros de la programmation mais vous pourrez apprendre différentes choses :

- Afficher quelque chose dans la console.
- Créer un script exécutable.
- Lancer un programme Python.

Ce sera la base donc pour toute la suite du cours ! Ce n'est clairement pas dur à maîtriser mais si vous n'êtes pas capable de faire ça, alors vous ne pourrez pas faire grand chose...

Ne tardons pas et commençons !

A partir de maintenant, lorsque du code sera donné, il le sera si possible en Python v3 ET en Python v2.6 ! Essayez de faire attention à la version que vous avez !

Hello World

Je ne vais pas passer par quatre chemins pour vous montrer la tête de notre premier programme et je vais vous le donner directement :

Code : Python - Shell Python v3

```
>>> print("Hello world!")
Hello world!
```

Code : Python - Shell Python v2.6

```
>>> print "Hello world!"
Hello world!
```

Quoi ?! C'est tout ?! C'est quoi ça ?! C'est nul !

Hola hola ! Essayez de taper la même chose dans votre shell pour commencer et vérifiez que cela fonctionne bien ! Maintenant, je vous explique un petit peu ce que l'on a fait là :

- **print()** : ceci est une fonction qui permet d'afficher dans la console ce que vous lui donner entre les parenthèses.
- **"Hello World!"** : ceci est une chaîne de caractères

Ici, on utilise donc la fonction *print()* qui permet d'afficher quelque chose pour demander d'afficher la chaîne de caractères *Hello World!* dans la console. Les fonctions seront vu plus en détail dans la suite du cours. Ce qu'il faut retenir maitenant, c'est que si vous voulez afficher quelque chose dans la console, il suffit d'utiliser cette ligne de code en remplaçant "Hello World!" par ce qui vous chante.

En Python v2.6, *print* est un mot clé et non pas une fonction. C'est une aberration corrigée dans la version 3 du langage... L'utilisation est cependant exactement la même, la différence est purement conceptuelle !

Si vous voulez vous amuser, vous pouvez essayer d'afficher toute sorte de chose dans la console.
Astuce 1 : Pour afficher plusieurs choses sur la même ligne, vous pouvez séparer ce que vous voulez afficher par des virgules.
Astuce 2 : Pour afficher quelque chose sur plusieurs lignes, utiliser plusieurs fois l'instruction *print ou utilisez le caractère*

spécial '\n' dans vos chaînes de caractères.

Créer un script

Un script est un fichier contenant un ensemble d'instructions. Dans notre cas, les instructions seront écrites en langage Python et lues par l'interpréteur pour les exécuter. Pour créer un script Python, vous pouvez utiliser n'importe quel éditeur de texte. Seulement, pour vous faciliter la tâche, il vaut mieux en utiliser un qui mette en couleur les mots clés (coloration syntaxique) utilisés par Python. Pour cela, vous avez le choix :

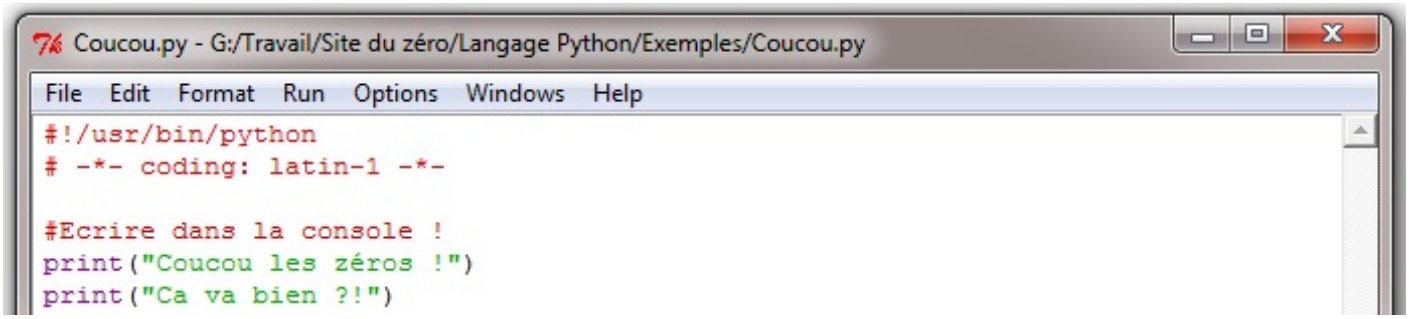
- **IDLE** : il vous propose un éditeur de texte avec la coloration syntaxique assez efficace et vous pourrez directement lancer vos scripts avec les raccourcis claviers
- **notepad++** : c'est un éditeur de texte particulièrement efficace qui gère la coloration syntaxique pour ce nombreux langages, il vous permet aussi de réduire les blocs de code pour vous permettre de travailler plus facilement (c'est mon chouchou)
- **nedit** : simple mais efficace, il gère aussi la coloration syntaxique pour de nombreux langages
- **PyPe** : c'est un environnement de développement qui vous permettra tout comme IDLE de lancer vos scripts. Il est muni d'un éditeur de texte à onglets, il gère la coloration syntaxique et propose aussi un shell.
- Et bien d'autres encore...

Tous ces éditeurs sont gratuits et peuvent être trouvés facilement sur Internet. Lorsque vous aurez acquis plus de compétences en programmation, vous pourrez trouver utile d'utiliser un environnement de développement (IDE) qui vous permet de travailler de façon très claire avec de nombreux fichiers, d'exécuter directement vos programmes et propose aussi le plus souvent un débugger qui permet de repérer facilement les problèmes dans votre code.

Si vous désirez maintenant écrire un fichier qui puisse être exécuté par l'interpréteur Python, celui-ci doit se terminer par l'extension **.py** qui indique que c'est un fichier de code Python.

Il vous arrivera peut-être de tomber sur des fichiers ayant pour extension .pyc. Il s'agit de fichier Python précompilés, vous ne pouvez donc plus voir le code source mais il sont exécutables.

Essayez de créer un fichier .py et d'y écrire du code Python en vous aidant du programme Hello World par exemple.
Vous devriez obtenir quelque chose qui ressemble à ça :



Vous pouvez voir ici, le code qui se colorie en fonction de ce que j'écris, ce qui indique que mon fichier Coucou.py a bien été reconnu comme un fichier de code Python :

- En rouge : les commentaires. Tout ce qui se trouve après le caractère # ne sera pas considéré par l'interpréteur, ce sont des commentaires pour vous et ceux qui liront le code. Il est très important de bien commenter son code afin qu'il soit lisible et facilement compréhensible.
- En violet : les mots clés du langages Python. Tous les mots clés que vous utiliserez seront mis dans cette couleur, ainsi vous vous rendrez compte tout de suite si votre code est bien reconnu.
- En vert : les chaînes de caractères. Tout ce qui se trouvera être entre deux " sera donc écrit en vert et ne sera pas considéré comme étant du code à exécuter mais quelque chose à afficher dans notre exemple.

La coloration syntaxique dépend de l'éditeur de texte que vous avez choisi et les couleurs peuvent donc être différentes suivant l'éditeur que vous utilisez. Ces couleurs ne sont là que pour vous faciliter la lecture de votre code et n'ont pas de valeur en elles-mêmes. Elles ne sont pas intrinsèques au langage Python. Vous verrez d'ailleurs que le site du zéro utilise son propre jeu de couleur pour la coloration syntaxique de Python.

En en-tête du fichier se trouvent des commentaires très particuliers. Les deux lignes suivantes :

Code : Python

```
#!/usr/bin/python
```

```
# -*- coding: latin-1 -*-
```

Ces deux lignes sont des commentaires avec une utilité particulière. La première permet de dire que c'est un fichier Python et cela rendra vos scripts exécutable par un simple double clic dessus par exemple. La seconde ligne précise l'encodage utilisé dans le fichier. Ici, l'encodage est *latin-1* qui m'autorisera à écrire les accents notamment pour les commentaires. Si vous ne précisez pas l'encodage, il se peut que vous ne puissiez utiliser que les caractères de base (aucun accent, pas de caractères spéciaux).

Depuis Python v3, l'encodage par défaut est UTF-8 qui vous permet donc d'utiliser à peu près tout ce que vous voulez !

Lancer un script

Maintenant que vous avez écrit votre premier script, il va falloir l'exécuter ! Il est possible de le faire de différentes manières.

Fichier exécutable

J'appelle le lancement direct le fait d'aller chercher directement le fichier que vous venez d'écrire dans votre navigateur et de le lancer en cliquant dessus comme vous lanceriez une application. Avec notre exemple, vous devriez voir apparaître et disparaître quasiment instantanément une console. En fait, c'est que votre programme s'est bien lancé et terminé.

Lancer en console

Si vous voulez voir un petit peu mieux ce que donne votre programme, vous devrez lancer une console (taper *cmd* dans le champ *Executer* sous Windows mais ce n'est pas le plus simple sous Windows...) et en vous plaçant dans le bon répertoire (celui qui contient votre fichier), vous pouvez taper :

Code : Console

```
python Coucou.py
```

Sous Windows, vous n'avez pas besoin de taper python. Seul le nom du script à exécuter est nécessaire.
Sous Linux, vous n'aurez pas non plus besoin de taper python si vous avez écrit *#!/usr/bin/python en en-tête de votre script.*

En faisant cela, le résultat de votre programme devrait alors s'afficher .

Lancer avec IDLE

Si vous voulez lancer votre script avec IDLE, il faut aller dans le menu Run et faire Run module (raccourci : F5). Votre programme se lancera alors tout seul dans le un shell IDLE. C'est la solution la plus simple pour un début !
Vous savez maintenant comment écrire quelque chose dans la console et comment lancer un programme Python... Vous êtes maintenant prêt à apprendre à programmer. Dans le prochain chapitre, vous apprendrez à stocker de l'information à l'aide de variables.

Les variables

Vous savez lancer un programme Python mais vous ne savez pas encore programmer. L'informatique, comme je l'ai expliqué un peu avant, c'est la science de l'information. Le but va donc être maintenant de travailler avec de l'information et de la stocker en mémoire.

C'est donc ce que nous allons apprendre à faire dans cette partie ! Ici, nous verrons :

- A quoi servent les variables
- Comment les utiliser
- Les différents types de données utilisables
- Les principaux mots clés de Python

Ne perdons pas de temps en bavardage et apprenons à programmer tout de suite !

Une variable, c'est quoi ?

Une variable, c'est un nom, une référence, qui pointe vers des données en mémoire. C'est aussi simple que ça. Et c'est donc à l'aide de nombreuses variables que vous pourrez stocker tout ce dont vous avez besoin pour vos programmes.

Ce qui consterne toujours un peu les jeunes programmeurs, c'est la quantité assez limitée de types de variables qui existent. En Python, les données que l'on peut utiliser sont des types suivants :

- **Les nombres entiers** (*integer* ou *long* en anglais suivant leur longueur)
- **Les nombres à virgule flottante** (*float*) : ce sont les nombres décimaux que l'on apprend à partir du CM1 si mes souvenirs sont bons !
- **Les chaînes de caractères** (*string*, celui qui rigole au fond de la classe, tu sors !)
- **Les booléens** (*boolean*) : qui vaudront forcément vrai (*true*) ou faux (*false*)

Il n'existe pas d'autres types de données que l'on peut utiliser ?! Mais je vais rien faire avec ça ! Comment je fais un programme avec des entiers et des chaînes de caractères ?!

Vous voyez, vous aussi vous êtes conterné... En effet, tous les programmes que vous utilisez sur votre ordinateur ne sont composés que de ces types de données ! Mais chaque programme peut en utiliser des milliers pour que vous puissiez apprécier votre musique préférée ou tout simplement lire ce tutoriel ! Tout ce que vous utiliserez sera composé de données de ces types, et rien d'autre ! Ne vous plaignez pas, il y a déjà plus choix qu'entre 0 et 1 !

Déclarer une variable

Maintenant que vous savez ce qu'est une variable, passons à la pratique ! On s'y met tout de suite avec la déclaration d'une variable :

Code : Python

```
peuh = 12
```

Ici, je déclare une variable appelée *peuh* et je lui affecte (signe =) la valeur 12. Vous voyez, c'est simple ! J'ai maintenant la valeur 12 qui est stockée en mémoire et je peux y faire appel à l'aide du nom *peuh*. Mais attention, il y a certaines règles à connaître lorsque l'on déclare une variable. La première est que la casse est prise en compte, ce qui veut dire que *peuh, Peuh, pEUh* et *PEUH* sont des variables différentes ! Vous ne pouvez pas non plus utiliser pour nom de variable les mots clés du langage Python. Ces mots ont tous une fonction bien particulière spécifique à la syntaxe du langage. En Python, les mots clés sont les suivants :

and	assert	break	class	continue	def	False
del	elif	else	except	exec	finally	None
for	from	global	if	import	in	
is	lambda	not	or	pass	print	
raise	return	try	while	yield	True	

Faites un essai dans le shell par exemple en y déclarant une variable et en tapant ensuite le nom de cette variable. Il devrait vous afficher la valeur de cette variable comme ceci :

Code : Python

```
>>> peuh = 12
>>> peuh
12
```

Ici, ma variable *peuh* est un entier (*integer*). En Python, le typage est dit dynamique. C'est-à-dire que le type des variables que vous déclarez sera donné automatiquement en fonction de la valeur que vous lui donnerez. Faisons un petit test dans le shell :

Code : Python

```
>>> peuh = 12
>>> peuh
12
>>> peuh = 'Peuh !'
>>> peuh
'Peuh !'
```

Je déclare ma variable en lui affectant la valeur 12. Si je l'affiche, alors c'est bien en entier de la valeur 12. Je lui affecte ensuite la valeur 'Peuh !' qui est une chaîne de caractères et lorsque je l'affiche, ma variable *peuh* est effectivement une chaîne de caractères, son type a été modifié automatiquement.

Les différents types des variables sont donnés par :

- **Les données numériques** qui seront du type *integer*, *long* ou *float* suivant le nombre que vous utiliserez
- **Les chaînes de caractères** délimitées par " " ou ' ' ou """ """ ou ''' '''
- **Les booléens** qui vaudront True ou False

Dans la suite du cours, vous verrez qu'il est possible d'utiliser des agrégats de données appelés liste et dictionnaire. Vous apprendrez un peu plus tard à les utiliser. En programmation par objet, vous apprendrez aussi à créer vos propres types de données, mais vous n'y êtes pas encore !

Si vous voulez rendre votre code lisible et compréhensible (car bientôt vous utiliserez des dizaines voire des centaines de lignes de code !), essayez d'utiliser des noms de variable explicites. J'utilise la normalisation suivante pour les noms de mes variables :

- Tous les noms de variables commencent par une lettre minuscule
- Les noms de variables ne sont composés que de lettres et de chifffres
- Chaque mot au sein d'un nom prend une majuscule (sauf le premier)

La norme à utiliser est libre mais doit être constante pour rendre le code plus lisible et facilement compréhensible.

Astuce : vous pouvez déclarer plusieurs variables en une seule ligne de code. Pour cela, regardez cet exemple :

Code : Python

```
a, b, c, d = 3, 23, 'Peuh !', True
```

Ici, sont déclarées quatre variables *a, b, c* et *d* avec respectivement les valeurs 3, 23, 'Peuh !' et *True*. Remarquez que les variables ainsi déclarées peuvent être de types tout à fait différents, cela ne pose aucun problème.

Utiliser des variables

Avoir des variables, c'est bien mais les utiliser, c'est mieux ! Regardons ce que l'on peut en faire avec un petit essai dans le shell.

Code : Python

```
>>> a, b, c, d = 3, 23, 5.78, 34e2
>>> a + b
26
>>> c
5.78
```

```
>>> c = a + b
>>> c
26
```

Voyons dans l'ordre ce que je fais :

- Je déclare quatre variables a, b, c et d et leur affecte les valeurs 3, 23, 5.78 et 34e2 (ceci est une notation scientifique qui signifie 34*10^2)
- Je demande de calculer le résultat de a + b : il me donne 26, ce qui est correct !
- Je lui demande d'afficher la valeur de c
- J'affecte la valeur a + b à la variable c
- J'affiche c et constate que sa valeur est bien devenue celle de a + b

Avec tout ceci, je m'aperçois donc que je peux utiliser mes variables dans des calculs ! Je peux donc faire des gros calculs en plusieurs fois en stockant des variables. Mais je peux aussi faire des opérations avec les chaînes de caractères (*string*) :

Code : Python

```
>>> peuh = "Peuh ! "
>>> peuh * 4
'Peuh ! Peuh ! Peuh ! Peuh ! '
```

Ainsi, je vous "peuhte" quatre fois en disant que je veux quatre fois ma chaîne de caractères !

Il existe aussi d'autres moyens d'affecter une valeur en effectuant une opération par la même occasion. Pour faire cela, vous pouvez utiliser les symboles suivants :

Symbole	Equivalent
a += 2	a = a + 2
a -= 2	a = a - 2
a *= 2	a = a * 2
a /= 2	a = a / 2
a **= 2	a = a ** 2
a //= 2	a = a // 2
a %= 2	a = a % 2

Avec ces opérations, vous pouvez ainsi vous éviter la répétition du nom de certaines variables ! Avant de passer à la suite, entraînez-vous à déclarer des variables et à faire des opérations en les utilisant. Vous pouvez aussi essayer de les afficher dans la console avec ce que l'on a vu dans le programme Hello World.
Bien, maintenant que vous savez déclarer des variables et les utiliser dans des calculs et l'affichage, on va passer à une partie un peu plus coriace où l'on va apprendre à contrôler le flux d'instructions. Ne paniquez pas, vous verrez, c'est simple !

Contrôle du flux (1/2)

Vous savez créer des variables et faire des calculs. C'est bien mais ça ne nous avance pas beaucoup ! Vous pouvez utiliser Python comme une grosse calculatrice mais c'est à peu près tout pour le moment. Dans cette partie, vous allez apprendre à contrôler le flux d'instructions. Nous allons donc voir ici :

- Les conditions : pour faire quelque chose ou pas selon la situation
- Les boucles : pour répéter des instructions

Cela paraît court dit comme ça, mais cette partie sera assez longue car c'est une des plus importantes de ce cours ! Et vous devrez maîtriser parfaitement tout ce qui se trouve dans cette partie car ce sera la base de tous vos programmes ! Au boulot !

Si ça... alors ça...

La première chose qui va vous permettre de gérer le flux d'instructions, ce sont les structures conditionnelles. Comment fait-on une structure conditionnelle ? Et bien voici un petit exemple qui va vous permettre de voir comment cela fonctionne :

Code : Python - v3

```
print("Coucou !")
unNombre = input("Donnez moi un nombre : ")
if unNombre < "50" :
    print("Votre nombre est plus petit que 50.")
elif unNombre == "50" :
    print("Votre nombre est exactement 50")
else :
    print("Votre nombre est plus grand que 50.")
```

Code : Python - v2.7

```
print "Coucou !"
unNombre = raw_input("Donnez moi un nombre : ")
if unNombre < "50" :
    print "Votre nombre est plus petit que 50."
elif unNombre == "50" :
    print "Votre nombre est exactement 50"
else :
    print "Votre nombre est plus grand que 50."
```

N'essayez pas d'entrer autre chose qu'un nombre pour le moment. Nous gèrerons les erreurs plus tard dans le cours.

Alors, que faisons-nous ici ? Voyons ligne par ligne les nouveautés.

Code : Python

```
unNombre = input("Donnez moi un nombre : ")
```

Ici, j'utilise la fonction *input* (tout comme *print*) qui permet à l'utilisateur d'interagir avec le programme. Cette ligne déclare donc une variable *unNombre*, écrit dans la console "Donnez moi un nombre : " et attend que l'utilisateur entre quelque chose dans la console pour donner une valeur à *unNombre*.

Code : Python - v3

```
if unNombre < 50 :
    print("Votre nombre est plus petit que 50.")
elif unNombre == 50 :
    print("Votre nombre est exactement 50")
else :
    print("Votre nombre est plus grand que 50.")
```

Ici, c'est la structure conditionnelle. Pour les non anglophones, je traduis les mots clés : si (*if*), autre si (*elif*, contraction de *else if*) et sinon/autre (*else*). Voyons plus en détail comment cette structure est construire. La première ligne :

Code : Python

```
if unNombre < "50" :
```

est composé de trois choses :

- **if** : cela marque le début de la structure conditionnelle
- **unNombre < 50** : ici, on effectue la comparaison de la variable *unNombre* avec le nombre 50. Le résultat de cette opération sera un booléen qui vaudra donc soit *True* (vrai) soit *False* (faux)
- **':'** : qui est le caractère qui marque la fin de la condition.

Notre ligne revient donc à dire "Si ma variable *unNombre* est inférieure à 50 alors...". Mais alors quoi me demanderez-vous ! La structure n'est pas terminée, il faut maintenant lui dire quoi faire si cette condition se vérifie. C'est ce que va faire la ligne suivante :

Code : Python

```
print("Votre nombre est plus petit que 50.")
```

Si la condition est vérifiée, alors je lui demande d'écrire quelque chose dans la console. Remarquez bien que cette ligne est indentée, c'est-à-dire qu'une marge marquée par un caractère tabulation (ou un nombre fixe d'espaces) va indiquer que cette ligne ne doit être exécutée uniquement si la condition a été vérifiée ! Cette ligne est appelée **un bloc de code** et pourrait en fait être composée d'autant de lignes de code que vous le souhaitez. Ca y est ! Vous contrôlez le flux d'instructions ! La suite :

Code : Python

```
elif unNombre == "50" :
    print("Votre nombre est exactement 50")
else :
    print("Votre nombre est plus grand que 50.")
```

est parfaitement facultative. Elle permet de compléter la condition en indiquant d'autres actions possibles si la première condition ne se vérifie pas. Ainsi, si le nombre n'est pas plus petit que 50, alors nous vérifions s'il est égale ou non à 50. Si oui, le bloc de code correspondant est exécuté. Le dernier cas marqué par *else* s'exécutera à chaque fois qu'aucune des conditions au dessus n'a été vérifiée.
En résumé, notre code en langage naturel donnerait :

Code : Autre

```
Ecrire "Coucou !"
Demander d'écrire un nombre.
Si ce nombre est plus petit que 50, écrire "Votre nombre est plus petit que 50."
Si ce nombre vaut 50, écrire "Votre nombre vaut exactement 50".
Si ce nombre ne correspond à aucune des conditions ci-
dessus, écrire "Votre nombre est plus grand que 50".
```

Les blocs de code

Petite parenthèse, voyons un peu comment fonctionnent les blocs de code en Python tant que c'est encore nouveau pour vous ! Rappelons déjà ce qu'est un bloc de code. Un bloc de code, de façon générale, c'est un ensemble d'instructions. En Python, il sera caractérisé par son indentation, c'est-à-dire le nombre de tabulations ou d'espaces en début de chacune des lignes qui le composent.

Il est possible d'imbriquer des blocs de code les uns dans les autres. On peut donc avoir quelque chose de la forme :

Ainsi, on peut effectuer des instructions dans un bloc, créer un nouveau bloc puis reprendre notre premier bloc. Si l'on convertit le schéma ci-dessus en exemple, on pourrait avoir quelque chose de la forme :

Code : Python - v3

```
print("Bloc de code principal !")
if trucMuche :
    print("Premier bloc de code !")
    if machinTruc :
        print("Second bloc de code !")
        if biduleChouette :
            print("Troisième bloc de code !")
        print("Retour dans le second bloc de code !")
    print("Retour dans le premier bloc de code !")
print("Retour dans le bloc de code principal !")
```

Ici, *trucMuche, machinTruc* et *biduleChouette* sont des variables de type booléen. On peut donc chaîner les blocs de code sans aucun problème ! C'est ce qui va vous permettre de contrôler votre flux d'instructions.

Maintenant, revenons en aux structures conditionnelles avec un petit cours de logique !

Un peu de logique

Revenons en à nos booléens et à nos si ! Un petit rappel de ce dont vous avez besoin pour faire une structure conditionnelle :

- Le mot clé *if*
- Un booléen
- Un caractère ':'

Euh, je fais comment des booléens déjà ? Et c'est quoi ?!

Alors un booléen, c'est une variable qui vaut soit *True* (vrai), soit *False* (faux). C'est un peu une sorte d'interrupteur si vous voulez, il est soit ouvert soit fermé. Et comme il existe des opérations pour créer des nombres, il existe des opérations pour créer des booléens qui donneront *True* si c'est vérifié et *False* sinon :

Symbole	Signification
==	égalité
!=	inégalité
>	supériorité stricte

>=	supériorité
<	infériorité stricte
<=	infériorité

Avec toutes ces opérations, vous pouvez donc comparer des données numériques (comme vous avez sûrement appris quand vous étiez à l'école primaire).

Il ne faut surtout pas confondre les opérations = et ==. Dans un cas, cela affecte une valeur à une variable, dans un autre, les valeurs de deux variables sont comparées !

Les opérateurs de supériorité ou d'infériorité fonctionnent aussi sur les chaînes de caractères en considérant l'ordre alphabétique.

Il existe ensuite trois opérateurs (sous forme de mots clés) pour modifier ou combiner des booléens :

- **and** (et) qui permet de multiplier les booléens.
- **or** (ou) qui permet d'additionner les booléens.
- **not** (non) qui permet d'inverser les booléens.

Pour mieux comprendre comment fonctionnent les booléens, considérez qu'ils prennent pour valeur 0 ou 1 (d'ailleurs, dans certains langages, les booléens n'existent pas et sont en fait représentés par 0 ou 1).
Pour ceux qui ne sont pas familiers avec l'utilisation des booléens, nous allons voir comment fonctionnent ces opérations. Ici, nous allons considérer que 0 est faux et que tout autre nombre est vrai (et c'est le cas en Python, si vous utilisez des nombres directement comme booléen, vous constaterez que seul 0 vaut *False*, tout le reste vaut *True*). Vous allez voir, tout cela a du sens.
On va traduire les opérations suivantes :

- *bool1* **and** *bool2* : *bool1* * *bool2*
- *bool1* **or** *bool2* : *bool1* + *bool2*

Maintenant, considérons les quatre cas possibles pour chacune de ces opérations :

Opération and/*	bool1 = 0	bool1 = 1
bool2 = 0	0 / False	0 / False
bool2 = 1	0 / False	1 / True

Opération or/+	bool1 = 0	bool1 = 1
bool2 = 0	0 / False	1 / True
bool2 = 1	1 / True	2 / True

Ces petits tableaux (aussi appelés tables de vérité) permettent de mieux comprendre dans quel cas ces opérations donneront un résultat vrai et dans quels cas elles donneront un résultat faux.

La priorité des opérations **and** et **or** sont respectivement les mêmes que pour * et + (*/and est prioritaire donc).
L'opérateur **not** ne s'appliquera qu'au booléen devant lequel il est placé et uniquement à celui-ci.

Avec ceci, vous êtes donc capable de faire de belles conditions pour vos programmes. Entraînez vous un petit peu en essayant les opérations sur des petites opérations simples.

Astuce : On pense souvent une phrase logique lorsque l'on réfléchit à un problème mais des fois, c'est l'inverse que l'on veut tester... Et bien il existe deux petites formules appelées loi de Morgan qui sont très simples :

- not (bool1 and bool2) est équivalent à : not bool1 or not bool2
- not (bool1 or bool2) est équivalent à : not bool1 and not bool2

Si vous n'en êtes pas convaincu, essayez de faire les tables de vérités pour en vérifier le résultat et voir que cela donne bien le même tableau.

Résumé

Cette partie était un peu longue et vous avez certainement appris beaucoup de choses. Je vous propose donc un petit résumé de comment construire les structures conditionnelles en Python avec de petits exemples des différents formes qu'elle peut prendre :

Code : Python

```
if monBooleen :
    instructions...
```

OU

Code : Python

```
if monBooleen :
    instructions...
else :
    instructions...
```

OU

Code : Python

```
if monBooleen :
    instructions...
elif unAutreBooleen :
    instructions...
else :
    instructions...
```

OU

Code : Python

```
if monBooleen :
    instructions...
elif unAutreBooleen1 :
    instructions...
elif unAutreBooleen2 :
    instructions...
  .
  .
  .

elif unAutreBooleenN-1 :
    instructions...
elif unAutreBooleenN :
    instructions...
else :
    instructions...
```

Vous devez être incollable sur les problèmes de logique maintenant ! 

Dans la deuxième partie consacrée au contrôle du flux d'instructions, vous apprendrez à répéter un ensemble d'instructions. Allez, hop hop hop, on avance !

Contrôle du flux (2/2)

En programmant, vous verrez que vous serez souvent amené à répéter une suite d'instructions. Seulement, qui aurait envie de recopier du code encore et encore dans le but de répéter cette suite d'instructions ? Dans la plupart des langages de programmation, il existe ce que l'on appelle des structures itératives ou boucles. Ce sont des instructions et plus particulièrement des structures qui permettent alors de répéter des actions selon certaines conditions. Ici, nous allons voir comment créer des boucles. Nous verrons les différents types de boucles qui existent en Python et leur utilité :

- Les boucles *while*
- Les boucles *for*

Ce chapitre ne sera pas de tout repos alors accrochez-vous bien et surtout lisez bien ce qui va suivre !

Tant que ...

Souvent en programmation, on veut effectuer une suite d'instructions tant que certaines conditions sont respectées. On peut le faire avec la structure conditionnelle vue au chapitre précédent. Seulement, des fois, il peut être utile de répéter cette suite d'instructions et de les "faire tant que" les conditions sont respectées. C'est exactement ce que fait la boucle *while* (tant que). Elle se construit de la sorte :

Code : Python

```
while monBooleen :
    instructions...
```

Nous avons donc ici une structure, c'est donc l'indentation qui permettra de déterminer les instructions à exécuter dans cette boucle. Si vous n'avez pas bien saisi ce qu'est l'indentation et son utilité, je vous conseille de relire le chapitre précédant.

Que va faire cette boucle ? Et bien, elle va exécuter les instructions qui la suivent tant que *monBooleen* sera à la valeur *True*. Ainsi, si vous lancez ce code d'exemple :

Code : Python

```
while True :
    print("Je suis une boucle infinie !")
```

Vous devriez lancer ce que l'on appelle une boucle infinie et votre console devrait se remplir de la phrase "Je suis une boucle infinie !". C'est-à-dire que la suite d'instructions ne pourra jamais se terminer...

Pour arrêter ce programme, souvenez-vous du raccourci clavier : **ctrl+C** !

Pour vous éviter ce genre de problèmes, il faut donc toujours s'assurer de pouvoir sortir d'une boucle (sauf si bien sûr, on ne souhaite pas en sortir, cela peut arriver !). Pour cela, on peut faire quelque chose de la forme :

Code : Python

```
unNombre = 0
while unNombre < 50 :
    unNombre += 1
    print("J'apprends à compter :",unNombre)
```

Essayez ce code afin de vous assurez qu'il incrémentera (ajoutera 1) *unNombre* tant que celui-ci sera inférieur à 50. De cette façon, on peut compter facilement un nombre d'opérations ou répéter une opération un nombre de fois voulu.

Mais dans la pratique, vous verrez que la boucle *while* s'utilise particulièrement lorsque l'on veut exécuter un nombre de fois inconnu à l'avance une instruction. Par exemple, si vous demandez à un utilisateur de vous entrer des données mais que vous ne savez pas quand il souhaite s'arrêter à l'avance mais uniquement lorsqu'il l'indique. Prenons un petit exemple :

Code : Python

```
print("Entrez des données. (Entrée vide pour arrêter).")
boucler = True
while boucler :
    entree = input()
    if entree == "" :
        print("Nous avons terminé !")
        boucler = False
    else :
        print("Vous avez entré :",entree)
```

Dans ce petit programme, on ne sait pas combien de données l'utilisateur souhaite entrer. Seulement, il faut pouvoir s'arrêter. J'ai alors défini une condition d'arrêt de la boucle en insérant une structure conditionnelle imbriquée dans la boucle. Ainsi, lorsque l'entrée est vide, le programme sort de la boucle et peut se terminer.

Mots clés : break et continue

Pour gérer les instructions au sein de vos boucles, il existe deux mots clés particulièrement utiles qui sont *break* (casser) qui arrête la boucle où qu'elle en soit et *continue* (continuer) qui passe à l'itération/pas de boucle suivant. On peut donc s'éviter l'utilisation d'un booléen pour marquer une sortie de boucle par exemple :

Code : Python

```
print("Entrez des données. (Entrée vide pour arrêter).")
while True :
    entree = input()
    if entree == "" :
        print("Nous avons terminé !")
        break
    else :
        print("Vous avez entré :",entree)
```

Ici, dès que l'utilisateur fera une entrée vide, la boucle sera "cassée" et le programme en sortira. Le fait d'utiliser ici l'instruction *break* exactement le même résultat que précédemment. Seulement, il vaut mieux éviter de l'utiliser car en plus du fait qu'il peut rendre le code moins facilement compréhensible, cela n'est pas très bon pour la performance de vos programmes ! A connaître mais à éviter donc.

Le mot clé *continue*, lui, permet de passer directement à l'itération suivante sans exécuter la suite du bloc de code situé dans la boucle ! Par exemple :

Code : Python

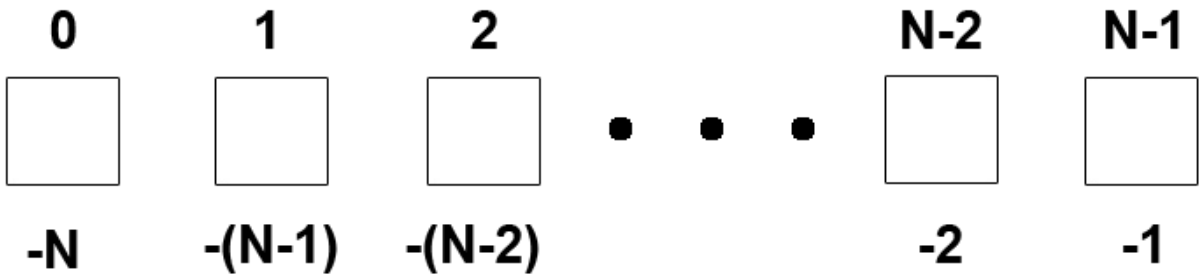
```
unNombre = 0
while unNombre < 50 :
     unNombre += 1
     if unNombre % 2 != 0:
          continue
     print("unNombre =",unNombre)
```

Ici, le code n'affichera *unNombre* que lorsque celui-ci sera pair (unNombre%2 = 0), lorsque celui-ci est impair, il passe au pas de boucle suivant.

Vous savez maintenant répéter une suite d'instructions. Nous allons voir qu'il existe un autre type de boucle qui s'applique aux objets itérables. Donc avant de voir cet autre type de boucle, définissons ce qu'est un objet itérable et voyons à quoi ils servent.

Les séquences

Tout d'abord commençons par la définition d'un objet itérable. C'est une séquence de données auxquelles il est possible d'accéder à l'aide d'un indice. Voyons ça avec l'aide d'un petit schéma d'une séquence de *N* éléments :



Une séquence est une suite de données placées en mémoire dans un ordre bien précis. Dans le schéma ci-dessus, on peut voir l'indice associé à chaque donnée dans la séquence.

Les indices commencent à 0 ! Ainsi, les indices d'une séquence de longueur N iront de 0 à N-1 !

Python est un des rares langages de programmation qui vous permet d'accéder aux éléments d'une séquence à l'aide d'indice négatif. Ainsi, -1 est associé au dernier élément de la séquence et -N au premier élément de la séquence. Mais alors, comment utilise-t-on les séquences en Python ? Il existe différents objets pour représenter des séquences, et il en existe un que vous connaissez déjà : les chaînes de caractères. En effet, ce sont des séquences ordonnées de caractères individuels. Et d'ailleurs, vous pouvez accéder aux caractères en particulier dans les chaînes de caractères via leur indice. Regardons un petit exemple dans le shell pour faire quelques essais :

Code : Python

```
>>> uneChaineDeCaracteres = "Peuh !"
>>> uneChaineDeCaracteres[0]
'P'
>>> uneChaineDeCaracteres[-1]
'!'
```

On voit ici que je déclare une variable *uneChaineDeCaracteres* et je demande ensuite le premier élément puis le dernier. Vous constaterez que le résultat semble plutôt correct ! Aussi, pour accéder à un élément d'une séquence, la syntaxe est la suivante :

Code : Python

```
sequence[indiceDeLElement]
```

Si vous cherchez à atteindre un élément avec un indice plus grand que la taille de votre séquence, vous aurez une erreur typique : *index out of range* . Cela vous indique que l'indice que vous utilisez est trop grand !

Maintenant que vous savez qu'une chaîne de caractères est une séquence, voyons un autre type de séquence très utile en Python : les listes. Vous en apprendrez bien plus sur ces séquences dans le chapitre qui y est consacré mais voyons déjà comment cela se déclare :

Code : Python

```
uneListe = ['Peuh !', 23, True, 28.2, 'Pouet !', False]
```

Ici, je déclare donc *uneListe* et lui affecte une suite de données. Ces données ne sont pas forcément toutes du même type ! Et maintenant, je peux obtenir ces données à l'aide de leur indice associé :

Code : Python

```
>>> uneListe = ['Peuh !', 23, True, 28.2, 'Pouet !', False]
```

```
>>> uneListe[1]
23
>>> uneListe[2]
True
```

Vous savez maintenant qu'il existe des objets dit itérables, sur lesquels on peut donc effectuer des procédés itératifs (des boules) et c'est ce que nous allons voir tout de suite avec la boucle *for*.

Pour chaque ... dans ...

Maintenant que vous commencez à connaître les séquences, une question doit vous venir à l'esprit : comment peut-on parcourir ces séquences ? Et bien c'est en cela qu'est utile la boucle *for*. Elle va vous permettre de répéter une suite d'instructions pour chaque élément de votre séquence. Voyons tout de suite un exemple de code :

Code : Python

```
uneSequence = "Peuh !"
for lettre in uneSequence :
    print("uneSequence contient la lettre :",lettre)
```

En langage naturel, on peut traduire ce petit code par :

Code : Autre

```
Je déclare une chaîne de caractères.
Pour chaque lettre dans cette séquence :
Ecrire "uneSequence contient la lettre :" et la lettre en question.
```

Ainsi, vous avez parcouru votre séquence et récupéré chaque élément de cette séquence dans la variable appelée *lettre*. La boucle for s'utilise donc de cette façon :

Code : Python

```
for variableDeRecuperation in sequence :
    instructions...
```

Vous définissez donc une variable dans laquelle sera stockée à chaque pas de la boucle la valeur d'une donnée dans la séquence. C'est simple non ? Vous verrez, c'est particulièrement pratique lorsque l'on travaille avec des ensembles de données !

Nous avons vu un peu plus avant comment répéter une suite d'instructions un nombre de fois limité. Ici, si vous connaissez à l'avance le nombre de fois que vous devez répéter vos instructions, vous pouvez utiliser aussi les boucles *for*. Pour cela, il vous suffit par exemple, de créer une liste avec une suite de nombres. Seulement, vous n'allez pas le faire à la main ! Les concepteurs de Python, comme tous les programmeurs, sont fainéants. Ainsi, ils ont créé une fonction, au même titre que *print* et *input*, qui permet de créer une liste correspondant à une suite de nombres : *range*(). Voyons un peu ce que fait cette fonction. Pour l'appeler c'est simple :

Code : Python

```
sequence = range(10)
```

Cette exemple créera alors une séquence équivalente à la liste : [0,1,2,3,4,5,6,7,8,9].

En version 3 de Python, cette fonction ne retourne pas une liste. Cette fonction crée un objet itérable organisé de type *range*, donc une séquence à ne pas confondre avec une liste qui est un type particulier de séquence !

Mais vous pouvez aussi l'utiliser directement dans la déclaration d'une boucle </attention>*for* comme ceci :

Code : Python

```
for i in range(10) :
    print("i =",i)
```

Vous voyez alors que *i* prend successivement les valeurs de 0 à 9.

En mathématique, on utilise souvent i, j, k ou l comme indice. Les noms explicites sont donc de rigueur en programmation mais lorsque l'on boucle sur des indices, ces petites lettres sont parfaitement compréhensibles. Ne soyez donc pas surpris de rencontrer ces notations !

Essayez donc de vous entraîner à utiliser les boucles avant de passer à la suite, mais si vous êtes impatient, le TP sera un excellent entraînement !
Bien, maintenant que vous connaissez les premières bases de la programmation, vous êtes fin prêt à faire de vous-même un programme. Ne vous inquiétez pas, je vais vous guider ! Il est donc temps de passer aux travaux pratiques !

TP 1 : Le Juste Prix

Vous y voilà, votre premier TP ! Ici, vous apprendrez à penser comme un programmeur et à vous débrouiller (pas trop quand même) pour faire votre premier programme tout seul !
Dans cette partie, nous verrons donc :

- Comment réfléchir à un problème
- Comment débugger un programme
- Un corrigé détaillé de l'exercice

Commençons sans plus attendre avec un jeu que tout le monde connaît : le jeu du Juste Prix !

Poser le problème

Le premier travail du programmeur est de bien poser le problème. Il est impossible de créer des programmes performants et/ou complexes si l'on ne réfléchit pas à comment on va le faire ! Et d'ailleurs, c'est une discipline à part entière appelée algorithmique. C'est la science des algorithmes.

Je n'ai pas la prétention ici de vous faire un cours d'algorithmique mais nous allons voir comment l'on peut réfléchir et construire notre programme avant même d'écrire du code ! Commençons par le commencement.

Enoncé du TP

Vous connaissez certainement ce jeu international appelé Juste Prix ! Et bien, dans ce jeu, lors de la grande finale, un candidat doit deviner la valeur d'un ensemble de cadeaux et s'il y arrive, il gagne le tout ! Alors, ce que nous voulons faire ici, c'est un petit jeu en console qui soit similaire. Les caractéristiques du programme sont donc :

- L'ordinateur doit générer un nombre aléatoire et nous proposer de deviner le montant.
- A chaque essai, l'ordinateur doit dire si c'est plus ou moins.
- Une fois le montant trouvé, il doit indiquer le nombre d'essais qu'il a fallu au joueur pour trouver la bonne réponse.

Ce n'est pas très compliqué. Mais vous verrez, vous deviendrez vite fan de ce jeu ! Si vous souhaitez un petit exemple de ce que cela peut donner en console, voici ce que j'obtiens avec mon programme :

Code : Autre - Juste Prix

```
-----------------
| Le Juste Prix |
-----------------
Choix de la difficulté :
1 - Entre 0 et 100
2 - Entre 0 et 1000
3 - Entre 0 et 10000
4 - Entre 0 et 100000
4
Le jeu commence, alors combien ?
50000
C'est plus !
75000
C'est plus !
87000
C'est moins !
81000
C'est moins !
78000
C'est moins !
76500
C'est moins !
75750
C'est moins !
75375
C'est plus !
75500
C'est plus !
75625
C'est plus !
75700
Bravo, vous avez trouvé la bonne réponse après 11 essais
```

Dans cet exemple d'interface, vous pouvez voir que je trouve la réponse assez vite. Hormis un coup de chance assez remarquable, j'utilise un algorithme appelé dichotomie qui permet de trouver une information dans une grande liste très rapidement. Pour l'appliquer, c'est simple, il suffit de couper l'intervalle des valeurs possibles en deux. Ainsi, je divise par deux les possibilités à chacune de mes réponses !

Penser l'algorithme

Bien, maintenant que nous avons l'énoncé de l'exercice, voyons ce que cet énoncé nous permet de définir sur notre algorithme et réfléchissons sur ce que nous utiliserons pour l'implémenter.

- On doit afficher quelque chose dans la console : nous utiliserons la fonction *print*.
- On doit générer un nombre aléatoire : vous ne pouvez pas le deviner puisque vous ne l'avez jamais vu, mais il existe une fonction et même plusieurs pour générer des nombres pseudo-aléatoires. Alors je vous donne les deux lignes qui vous seront utiles. La première est à placer au début de votre code et permet de pouvoir utiliser la fonction qui génère des nombres aléatoires (pour plus d'informations sur les nombres aléatoires en informatique, je vous conseille la lecture de l'excellent tutoriel de Natim et Sebsheep à ce sujet !).

  Code : Python

  ```
  from random import randint
  ```

  La deuxième est celle qui permet de générer le nombre aléatoire :

  Code : Python

  ```
  nombreAleatoire = randint(0, 100)
  ```

  Cette fonction génèrera un nombre entier aléatoire compris entre 0 et 100. Si vous voulez changer les bornes, il vous suffit de changer les valeurs entrées en paramètres dans la fonction.
- On veut pouvoir intéragir avec l'utilisateur, nous utiliserons donc la fonction *input*.
- La fonction *input* vous renverra une chaîne de caractères, il sera donc utile de la convertir en entier et Python est capable de le faire assez facilement. Dans les deux lignes qui suivent, il est demandé à l'utilisateur d'entrer quelque chose et la variable ainsi affectée est ensuite convertie en nombre à l'aide de la fonction *eval* (contraction de *evaluate*).

  Code : Python

  ```
  reponse = input()
  reponse = eval(reponse)
  ```

Pour ceux qui codent en Python v2.7, la fonction *input* existe aussi mais elle ne fait pas exactement la même chose. En effet, si l'utilisateur entre un nombre, elle donera alors directement un nombre... L'équivalent de la fonction *input* de la version 3 en Python v2.7 est *raw_input*. Dans la pratique, seule cette dernière fonction est utilisée, c'est pour cela qu'elle a été supprimée dans la version 3.

- Le nombre d'essais effectués par l'utilisateur est, *a priori*, inconnu. Il faudra donc répéter une suite d'instructions un nombre de fois indeterminé. C'est donc une boucle *while* qui semble être la réponse à ce genre de problème.
- L'ordinateur doit indiquer au joueur si le montant à deviner est plus grand ou plus petit que la réponse donnée, c'est donc une structure conditionnelle qui sera utilisée.

Vous voyez ? Un énoncé très court nous a donné une quantité d'informations très importante ! Nous avons ici tous les éléments qui vont composer notre programme. Donc à vous de jouer ! Allez, hop hop hop, au travail !

Débugger

Lorsque vous commencerez à programmer et même quand vous aurez atteint un niveau d'expert, vous n'écrirez certainement jamais un code parfait ! C'est parfaitement humain de faire des erreurs de syntaxes ou de réflexions. Ainsi, il va falloir corriger votre code et retirer tous les bugs qui s'y trouvent. Heureusement, des outils et un peu de méthode permettent de trouver facilement vos erreurs.

Pour toutes les erreurs de syntaxe, c'est-à-dire une erreur d'écriture d'une structure ou d'une affectation de variable, etc... l'interpréteur vous écrira toujours une erreur dans la console ! Il est là pour ça ! Et comme il est sympathique, il vous indique même où se trouve l'erreur dans votre code.
Une erreur dans la console est composée de :

- La ligne qui n'est pas bonne.
- Le numéro de la ligne et le nom du fichier dans lequel elle se trouve.
- Le type d'erreur (fin de ligne inattendue, caractère non reconnu, etc...)

Vous verrez plus tard d'ailleurs qu'il peut même afficher la suite d'instructions qui ne fonctionne pas pour vous permettre de mieux cibler l'erreur.
Les erreurs les plus courantes que vous commettrez sûrement par inadvertance sont les suivantes :

- Oubli du caractère ':' dans la définition d'une structure.
- Mauvaise indentation
- Utilisation de '=' à la place de '==' et inversement.

En cas d'oubli du caractère ':', il se peut que l'interpréteur ne vous indique par la bonne ligne et vous dise qu'une instruction n'est pas correcte. Pensez donc bien à vérifier que vous n'avez rien oublié dans la définition de vos structures si la ligne que vous voyez dans l'erreur vous semble parfaitement correcte.

Ces erreurs sont facilement évitables en relisant votre code mais aussi avec l'expérience. La syntaxe deviendra naturelle avec la pratique et vous ferez de moins en moins de fautes d'étourderie.

L'autre type d'erreur est là entièrement votre faute et correspond à un problème de raisonnement. Les erreurs les plus courantes sont :

- Un indice trop grand dans une liste (*index out of range*) : faites attention à ne pas utiliser un indice plus grand que la taille de votre liste lorsque vous voulez accéder à un élément...
- Une boucle infinie : la condition de sortie de la boucle n'est jamais vérifiée... Cela se passe le plus souvent lorsque que votre programme semble tourner dans le vide... Il n'y a qu'une seule solution à ça : **ctrl + C**. Cette commande "tuera" votre programme et vous permettra de le relancer sans faire souffrir inutilement votre ordinateur. Pensez donc bien à pouvoir sortir de vos boucles (*while*).
- Une condition qui ne se vérifie pas : essayez de poser votre problème logique en langage naturel. Traduisez ensuite vos phrases logiques en opérations. Et n'oubliez pas les lois de Morgan (cf chapitre Contrôle du flux (1/2)).

Voilà les principales erreurs que vous rencontrerez. Il n'y en a pas tant que ça mais vous verrez que le débuggage est en réalité la partie la plus longue du travail du programmeur !

Astuce : Un bon moyen de débugger est de regarder la valeur de vos différentes variables et de suivre le flux d'instructions. Pour cela, des outils appelés débuggers existent pour la plupart des langages. Malheureusement, ces outils sont souvent difficiles à utiliser et demandent une certaine expérience. C'est pourquoi, vous pouvez utiliser la méthode primitive qui a fait ses preuves : utiliser *print* ! Ainsi, vous pouvez vous mettre des marqueurs en indiquant dans la console quelle partie du code est en train d'être exécutée mais aussi la valeur de certaines variables qui vous intéressent. C'est un peu sale mais rapide et efficace ! Bien sûr, pensez à retirer tous ces *print* de votre code une fois le débuggage terminé !

Corrigé et amélioration

Bon, j'espère que vous avez réussi à faire quelque chose de potable sans avoir trop d'erreurs ! Maintenant, essayez tout de même de voir mon corrigé, peut-être verrez vous des choses intéressantes. Je vais essayer de l'expliquer en détail.

Secret (cliquez pour afficher)

Code : Python

```
# -*- coding: latin-1 -*-
########################
# Juste Prix #
# ------------------- #
# Créé : le 29/06/2010 #
# Auteur : Suzy #
########################

from random import randint

#Afficher l'interface :
print("-----------------\n" +
```

```
            "| Le Juste Prix |\n" +
            "-----------------\n" +
            "Choix de la difficulté :\n" +
            "1 - Entre 0 et 100\n" +
            "2 - Entre 0 et 1000\n" +
            "3 - Entre 0 et 10000\n" +
            "4 - Entre 0 et 100000")

    #Laisser le choix de la difficulté à l'utilisateur :
    difficulte = input()
    while (difficulte != "1" and difficulte != "2" and
           difficulte != "3" and difficulte != "4") :
        print("Veuillez entrer quelque chose de correct !")
        difficulte = input()

    #Initialiser les variables :
    nbEssais = 0
    reponse = -1
    #Générer un nombre aléatoire :
    if difficulte == "1" :
        aDeviner = randint(0, 100)
    elif difficulte == "2" :
        aDeviner = randint(0, 1000)
    elif difficulte == "3" :
        aDeviner = randint(0, 10000)
    else :
        aDeviner = randint(0, 100000)

    #Indiquer que le jeu à commencé :
    print("Le jeu commence, alors combien ?")

    #Boucle principale :
    while reponse != aDeviner :
        nbEssais += 1
        #Vérifier qu'un nombre est bien donné :
        reponseEstUnNombre = False
        while not reponseEstUnNombre :
            reponse = input()
            try :
                reponse = eval(reponse)
                reponseEstUnNombre = True
            except :
                print("Vous devriez donner un nombre...")
        #Dire si c'est plus ou moins :
        if reponse < aDeviner :
            print("C'est plus !")
        elif reponse > aDeviner :
            print("C'est moins !")

    #Terminer le jeu :
    print("Bravo, vous avez trouvé la bonne réponse après",nbEssais,
    "essais !")
```

Il n'existe jamais qu'une solution en programmation. La version que je propose n'est pas parfaite et n'est qu'une solution parmi d'autres. Sur un programme comme celui-ci, la performance générale n'est pas perceptible et la seule chose qui compte est ce que l'utilisateur, lui, perçoit ! Ne soyez donc pas découragé si votre programme n'est pas identique à mon corrigé !

Alors, voyons les choses dans l'ordre :

- **Lignes 1 à 7** : Je précise l'encodage de mon fichier (je peux donc utiliser les accents dans mes commentaires sans aucun problème par exemple) puis je documente mon script en précisant son nom, sa date de création et son auteur, moi )
- **Ligne 9** : J'importe la fonction qui permet de générer des entiers aléatoires.
- **Lignes 12 à 19** : J'affiche une petite interface textuelle dans la console avec un choix à faire pour l'utilisateur. Notez ici que je n'utilise qu'une seule fois *print* ! Je ne vous l'ai pas dit mais il existe un caractère spécial qui indique un retour à la ligne, c'est le *\n* que vous voyez. En utilisant ce caractère, je fais donc l'équivalent de plusieurs *print*. De même, tant que

je suis à l'intérieur d'une parenthèse, les espaces ou tabulations ne sont pas considérés comme des indentations délimitant des blocs de code. Je suis donc libre d'aligner mon code à ma guise pour le rendre plus lisible.

- **Lignes 22 à 26** : Je demande à l'utilisateur de choisir une difficulté à l'aide de la fonction *input()*. Afin que l'utilisateur ne fasse pas bugger mon programme par simple cruauté envers sa machine, je décide de vérifier que sa réponse est correcte. Comme je ne connais pas le degré de stupidité et/ou de sadisme de l'utilisateur, j'utilise une boucle *while* qui me permet de gérer sa bêtise à l'infini ! S'il ne donne pas une réponse correspondant à l'un des choix que je lui laisse, je lui indique qu'il n'a pas répondu quelque chose de correct et je lui redemande son choix.
- **Lignes 29 à 39** : Je déclare des variables pour stocker le nombre d'essais que l'utilisateur à fait, la réponse qu'il donne (que j'initialise à -1 afin d'être sûr qu'elle ne sera pas égale par inadvertance au nombre à deviner) et enfin, en fonction de la difficulté choisie, je génère un nombre entier aléatoire.
- **Ligne 42** : J'indique que le jeu a commencé.
- **Ligne 45** : Je commence ma boucle principale. Elle doit continuer tant que l'utilisateur n'a pas trouvé la bonne réponse.
- **Ligne 46** : J'incrémente le nombre d'essais du joueur.
- **Lignes 48 à 55** : Je crée ici une boucle pour gérer une entrée qui ne serait pas un nombre correct. Ainsi, si l'utilisateur rentre n'importe quoi, je ne pourrai pas effectuer la comparaison de sa réponse avec le nombre à deviner. Pour ne pas que cela arrive, j'utilise le bloc de code *try* qui me permet d'essayer de faire quelque chose et que ça puisse ne pas fonctionner sans que mon programme plante ! Vous verrez comment utiliser ce type de bloc de code dans le chapitre consacré aux exceptions.
- **Lignes 57 à 60** : Je dis au joueur si le nombre à deviner est plus grand ou plus petit que celui qu'il a donné. Attention, ici, il ne faut pas oublier que l'on sort de la boucle principale s'il donne la bonne réponse, il n'est pas nécessaire en toute logique de tester le cas d'égalité !
- **Ligne 63** : Je suis sorti de la boucle principale, le joueur a donc trouvé la bonne réponse. Je lui affiche alors en combien d'essais il l'a trouvée.

Pour ceux d'entre vous qui coderaient en Python v2.7, voici un corrigé :

Secret (cliquez pour afficher)

Code : Python - v2.7

```
# -*- coding: latin-1 -*-
#########################
# Juste Prix #
# -------------------- #
# Créé : le 29/06/2010 #
# Auteur : Suzy #
#########################

from random import randint

#Afficher l'interface :
print "-----------------\n"
print "| Le Juste Prix |\n"
print "-----------------\n"
print "Choix de la difficulté :\n"
print "1 - Entre 0 et 100\n"
print "2 - Entre 0 et 1000\n"
print "3 - Entre 0 et 10000\n"
print "4 - Entre 0 et 100000"

#Laisser le choix de la difficulté à l'utilisateur :
difficulte = raw_input()
while (difficulte != "1" and difficulte != "2" and
       difficulte != "3" and difficulte != "4") :
    print "Veuillez entrer quelque chose de correct !"
    difficulte = raw_input()

#Initialiser les variables :
nbEssais = 0
reponse = -1
#Générer un nombre aléatoire :
if difficulte == "1" :
    aDeviner = randint(0, 100)
elif difficulte == "2" :
    aDeviner = randint(0, 1000)
elif difficulte == "3" :
    aDeviner = randint(0, 10000)
```

```
else :
    aDeviner = randint(0, 100000)

#Indiquer que le jeu à commencé :
print "Le jeu commence, alors combien ?"

#Boucle principale :
while reponse != aDeviner :
    nbEssais += 1
    #Vérifier qu'un nombre est bien donné :
    reponseEstUnNombre = False
    while not reponseEstUnNombre :
        reponse = raw_input()
        try :
            reponse = eval(reponse)
            reponseEstUnNombre = True
        except :
            print "Vous devriez donner un nombre..."
    #Dire si c'est plus ou moins :
    if reponse < aDeviner :
        print "C'est plus !"
    elif reponse > aDeviner :
        print "C'est moins !"

#Terminer le jeu :
print "Bravo, vous avez trouvé la bonne réponse après",nbEssais,
"essais !"
```

Les codes sont relativement similaires, les deux seules différences étant la fonction *print* et la fonction *input*...

Je vous ai promis quelques petites idées d'améliorations alors chose promise, chose due ! Voici une petite liste d'améliorations que vous pouvez apporter à votre programme :

- Proposer de rejouer à l'utilisateur.
- Afficher les statistiques du joueur sur ses parties (combien de parties, dans quelle difficulté, moyenne d'essais, etc...)
- Faire que ce soit vous qui donniez un nombre et à l'ordinateur de le trouver !

Bravo ! Vous avez passé le premier cap et vous êtes maintenant un programmeur néophyte officiel ! Ah bah oui... Vous n'êtes pas encore amateur, vous avez encore tant de choses à apprendre !

Maintenant que vous connaissez les bases pour écrire du code Python, passons à la suite avec un peu de structuration du code avec les fonctions. Vous allez voir, c'est coriace, mais vous ne pourrez plus vous en passer !

Partie 2 : Bases et programmation fonctionnelle

Vous avez réussi à faire de vous-mêmes un petit jeu. Mais finalement, il vous reste énormément de choses à voir ! Dans cette partie, vous apprendrez les bases avancées du langages Python avec la gestion des fonctions et quelques concepts qui me semblent importants. Cette partie se consacre donc à la programmation dite fonctionnelle. Vous créerez vos propres fonctions et y ferez appel.

Préludes aux fonctions

Vous y êtes presque avec les bases de Python ! Dans cette partie, vous allez en apprendre un peu plus sur ce qu'est une fonction et à quoi cela peut bien servir.
Vous verrez aussi qu'il existe de nombreuses fonctions prédéfinies qui peuvent être particulièrement utiles pour toute sorte d'applications !

Encore une fois, ne perdons pas trop de temps en bavardage et mettons nous au travail !

Qu'est-ce qu'une fonction ?

Commençons par définir ce que l'on appelle une fonction en programmation. Vous en avez déjà utilisé jusqu'à maintenant et vous savez certainement que c'est un morceau de code que l'on appelle et qui va effectuer une suite d'actions. Les choses importantes qui vont composer une fonction sont :

- **Son nom** : cela vous permettra de l'appeler.
- **Ses entrées** : ce sont des données/variables que vous lui passerez qui sont appelées paramètres ou arguments. En python, ces arguments sont placés entre parenthèses.
- **Sa sortie** : ce que la fonction vous retournera comme résultat qui pourra être stocké dans une variable.

Pour appeler une fonction, on a alors un code du type :

Code : Python

```
variablePourStockerLaSortie = nomDeLaFonction(argument1, argument2,
...)
```

Finalement, une fonction, c'est un peu une boîte noire. Tout ce qui vous importe c'est ce que vous lui donnez en entrée et ce qu'il en ressort. Les moyens mis en oeuvre à l'intérieur de celle-ci n'ont pas besoin d'être connus. Si l'on prend l'exemple de *print*, vous savez que vous lui passez en argument des données à écrire et que cela vous les écrit. Mais vous n'avez pas besoin de savoir comment cela se fait ! Cette fonction ne vous retourne d'ailleurs aucun résultat.

De même si l'on étudie la fonction *input*. On voit que l'on connaît son nom, *input*. On peut aussi choisir de lui passer en argument une chaîne de caractères ou une suite de données à afficher. Le résultat est que cela sera affiché, que l'utilisateur devra entrer quelque chose et que cette entrée sera retournée pour être stockée dans une variable comme dans le code suivant :

Code : Python

```
variable = input("Je suis un argument à écrire en console !")
```

Vous verrez au fur et à mesure du cours que les fonctions peuvent accepter des arguments facultatifs, c'est-à-dire que vous n'êtes pas obligé de leur donner une valeur. De même, une partie de ces arguments facultatifs peut être appelée dans le désordre du moment que vous spécifier le nom des arguments. Schématiquement, cela donne :

Code : Python

```
variable = fonction(argument1, argument2, nomArgument = valeur,
nomArgument2 = valeur2, ...)
```

Vous allez voir qu'il existe en Python de nombreuses fonctions intégrées que vous serez amené à utiliser régulièrement.

Fonction intégrées et importées

Les fonctions intégrées sont des fonctions que vous pouvez appeler dans n'importe quel code Python et qui seront toujours

parfaitement définies. Vous en avez déjà utilisé. *print, input* et *eval* sont des fonctions intégrées (*built-in functions* en anglais). Mais vous allez voir que vous pouvez aussi importer des fonctions existantes utiles pour toute sorte d'applications.

Fonctions intégrées

Une liste non exhaustive de fonctions intégrées :

- **print** : permet d'écrire dans la console.
- **input** : permet de récupérer des entrées au clavier.
- **eval** : évalue la valeur numérique d'une chaîne de caractères correspondant à un nombre ou une suite d'opérations mathématiques.
- **range** : retourne une séquence de nombres entiers compris entre 0 et le nombre passé en argument.
- **min/max** : retourne le minimum/maximum dans les élements de la séquence passée en argument.
- **len** : retourne la longueur d'une séquence passée en paramètre.
- Et bien d'autres...

Pour trouver la liste complète des fonctions intégrées au langages Python, vous pouvez aller lire la documentation au lien suivant :

http://docs.python.org/py3k/library/functions.html

Quelque soit le langage de programmation que vous utiliserez, la documentation sera en anglais. Je vous avais dit que vous deviez vous y mettre, il est temps de vraiment commencer !

Sur cette page, vous trouverez absolument toutes les fonctions intégrées au langage. Pour chacune d'elle, vous avez son nom et entre parenthèses les arguments qu'elle peut accepter. Le tout est accompagné d'un petit texte explicatif avec quelques fois des exemples lorsque la fonction est un peu complexe.

Si ce genre de page vous rebute, vous n'irez pas très loin en programmation car c'est souvent sur ce type de page que vous passerez votre temps afin de comprendre comment utiliser telle ou telle fonction ! Il n'y a pas de méthode particulière pour lire la documentation... Il suffit de lire, de tester par soi-même pour bien saisir le fonctionnement de ce que l'on utilise et c'est tout. Les forums de développeurs pourront aussi vous apporter des quantités d'informations utiles. Il est rare que vous ayez un problème que personne n'ait jamais rencontré !

Importer des fonctions

Sur la page de documentation que je vous ai donnée, vous ne trouverez que les fonctions intégrées directement à Python. Mais il en existe bien d'autres ! Des milliers d'autres ! Et vous pouvez toutes y avoir accès sans rien installer de plus sur votre ordinateur ! Pour cela, il suffit de les importer. Souvenez-vous, dans les travaux pratiques sur le Juste Prix, je vous avait donné deux lignes de codes à utiliser pour générer des nombres pseudo-aléatoires. La première d'entre elles était :

Code : Python

```
from random import randint
```

Cette ligne importe la fonction *randint* afin de la rendre utilisable dans notre code. On y trouve le mot clé *from* suivi du nom du module *random*. Le mot clé *from* permet d'indiquer un nom de module dont on veut importer les fonctions. Le mot clé *import* permet quand à lui d'importer des fonctions particulières, ici *randint*. Littéralement, la ligne ci-dessus pourrait se traduire par : Du module *random*, importer la fonction *randint*.

Pour importer des fonctions, on peut alors utiliser la forme suivante :

Code : Python

```
from nomDuModule import nomDeLafonction1, nomDeLafonction2, ...
```

Ainsi, vous pouvez importer en une seule ligne plusieurs fonctions provenant d'un même module en séparant les noms de fonction par des virgules. Si vous faites ainsi, vous pouvez utilisez directement les fonctions importées en utilisant leur nom au même titre que *print* ou *input*.

Vous pouvez importer toutes les fonctions d'un module d'un seul coup en remplaçant les noms de fonction par un symbole *. Ainsi, vous n'aurez pas besoin de connaître le nom de toutes les fonctions que vous désirez importer ni de les taper... Exemple :

Code : Python

```
from nomDuModule import *
```

Il existe une autre manière d'importer des fonctions :

Code : Python

```
import nomDuModule
```

En effectuant une telle importation, vous importez en réalité un module dans son ensemble et vous ne pourrez pas faire appel aux fonctions simplement avec leur nom. Pour y faire appel, vous devrez préciser le module auxquelles elles appartiennent. Pour appeler une fonction du module que vous avez importer, vous devrez alors taper quelque chose de la forme suivante :

Code : Python

```
sortie = nomDuModule.nomDeLaFonction(arguments)
```

Cela est particulièrement utile lorsque plusieurs modules que vous importez possèdent des fonctions du même nom. Cela peut arriver plus souvent qu'on ne le pense. De cette manière, vous levez toutes les ambiguïtés qu'il pourrait y avoir et vous vous éviterez quelques crises de nerfs !

Si vous voulez savoir ce qu'il vous est possible de faire avec les modules de base en Python, un chapitre y est consacré dans la suite du cours. Sinon, si vous vous sentez l'âme d'un aventurier anglophone, vous pouvez toujours vous lancer directement dans la documentation officielle complète :

Python v3 :
http://docs.python.org/py3k/
Python v2 :
http://docs.python.org/

Exemple : manipuler des fichiers

Maintenant que vous avez mieux compris ce qu'est une fonction, nous allons voir un bel exemple de fonctions intégrées qui permettent de manipuler des fichiers ! Vous en aurez besoin pour lire des fichiers mais aussi pour enregistrer des données par exemple.

Ouvrir un fichier

Pour pouvoir manipuler un fichier, il faut d'abord en ouvrir un ! Pour cela, une fonction intégrée, *open*, permet d'ouvrir un fichier. Elle demande deux arguments : le nom du fichier (ou le chemin vers ce fichier) à ouvrir et le mode de lecture. On a donc quelque chose de cette forme :

Code : Python

```
fichier = open(nomDuFichier, modeDeLecture)
```

Cette fonction retourne un objet de type fichier auquel on pourra appliquer des fonctions que nous allons voir. Il est ouvert dans le mode de lecture spécifié. Mais qu'est-ce qu'un mode de lecture ? C'est simple ! Cela indique si vous voulez lire et/ou écrire le fichier. Les principaux modes de lectures sont les suivants :

- **"r"** comme *read* (lire) : ouverture en lecture uniquement (le fichier doit exister !)
- **"w"** comme *write* (écrire) : ouverture en écriture uniquement (crée le fichier, l'efface préalablement s'il existe déjà)
- **"a"** comme *append* (ajouter) : ouverture en mise à jour (crée le fichier, écrit à partir de la fin du fichier s'il existe déjà)

- **"r+"** : ouverture en lecture et écriture
- **"w+"** : ouverture en lecture et écriture (a le même effet que 'w')
- **"a+"** : ouverture en lecture et écriture (a le même effet que 'a')

Vous voyez donc qu'il est possible de lire et/ou d'écrire un fichier suivant ce que l'on veut en faire. Le mode de lecture seule vous évite notamment de modifier un fichier par inadvertance...

Par défaut, si vous ne précisez pas de mode de lecture, le fichier sera ouvert en lecture uniquement.

Le chemin spécifié pour le fichier prend sa racine dans le dossier d'exécution du programme. Si vous voulez charger un fichier dans un sous dossier, il faut donc le préciser. Par exemple, si vous avez un dossier "textes" dans lequel vous voulez lire un fichier appelé "Peuh.txt", pour l'ouvrir il faudra utiliser le code suivant :

Code : Python

```
fichier = open("textes/Peuh.txt", "r")
```

Dans un soucis de compréhension, même si vous ne souhaitez que lire le fichier, essayez de toujours spécifier le mode de lecture. Cela permet d'éviter les ambiguïtés !

"../" remontera dans le dossier parent. Ces deux points peuvent être utilisés n'importe où dans le chemin d'accès à un fichier et constitue le nom générique pour désigner le dossier parent.

Ecrire un fichier

Maintenant que vous savez comment ouvrir un fichier, nous allons voir comment écrire dans un fichier. C'est la fonction *write* appliquée à un fichier qui va permettre de faire ça. Mais aussi, lorsque l'on ne veut plus rien faire avec un fichier, il faut penser à le fermer à l'aide de la fonction *close*. Pour écrire dans un fichier, nous utiliserons donc quelque chose dans ce style :

Code : Python

```
fichier = open("test.txt","w")
fichier.write("Peuh ! J'essaye d'écrire dans un fichier !")
fichier.close()
```

Remarquez ici que les fonctions *write* et *close* sont directement appliquées à la variable fichier. Cela indique donc que les opérations que l'on demande d'effectuer doivent se faire dans ce fichier là et pas un autre.

Si vous oubliez de fermer votre fichier avant la fin de votre programme, il se peut que celui-ci ne soit pas sauvegardé ! Vous ne pourrez donc pas y trouver ce que vous vouliez y écrire !

Vous pouvez appliquer plusieurs fois la fonction *write* à un même fichier. Dans une boucle par exemple, pour écrire une suite de données calculées ou issues d'une séquence. Par exemple :

Code : Python

```
fichier = open("test.txt","w")
for indice in range(30) :
    fichier.write("Je suis la ligne " + str(indice + 1) + " !\n")
fichier.close()
```

Si vous regardez le fichier qui résulte de l'exécution de ce programme, vous verrez 30 lignes écrites. J'utilise ici une fonction intégrée que vous ne connaissez pas : *str*. Cette fonction permet de changer le type d'une variable pour la transformer en chaîne de caractères. Il en existe pour tous les types de bases :

- *int* : convertit en *integer* (entier)
- *float* : convertit en *float* (nombre décimal, à virgule flottante)

- *bool* : convertit en *boolean* (booléen)
- *str* : convertit en *string* (chaîne de caractères)

Si vous utilisez ces fonctions, vous devez vous assurer que la conversion est effectivement possible. En outre, seule la conversion des types de base en chaîne de caractères est toujours possible !

Dans le chapitre suivant consacré aux chaînes de caractères, vous apprendrez à formater les chaînes de caractères. Cela vous évitera d'utiliser les fonctions de conversion de type qui peut rendre le code assez vite illisible...

Pour écrire dans un fichier, il existe aussi la fonction appelée *writelines* (écrire lignes) qui permet d'écrire tous les éléments d'une liste dans un fichier. Le code suivant :

Code : Python

```
liste = ["Peuh !\n","Alors, heureux ?!\n", "Allez, cassos !\n"]
fichier = open("test.txt","w")
fichier.writelines(liste)
fichier.close()
```

Malheureusement, contrairement à ce que pourrait laisser penser le nom de la fonction, *writelines* n'écrit pas les éléments de la liste passée en paramètre sur plusieurs lignes. C'est pourquoi je rajoute ici le caractère de fin de ligne à la fin de chacune des chaînes de caractères dans ma liste.

Lire un fichier

Pour lire un fichier, vous devez d'abord l'ouvrir, et ensuite la méthode *readline* (lire ligne) retournera tout d'abord la première ligne. Puis si vous la rappelez, la seconde et ainsi de suite jusqu'à la fin du fichier. Par exemple, si l'on reprend le fichier précédemment créé et qu'on exécute ce code :

Code : Python

```
fichier = open("text.txt","r")
print(fichier.readline())
print(fichier.readline())
fichier.close()
```

Il devrait s'afficher :

Code : Console

```
Peuh !
Alors, heureux ?!
```

dans la console. Mais un fichier est en fait une séquence, c'est-à-dire un objet itérable ! Vous pouvez donc facilement boucler sur de tels objets pour en lire toutes les lignes. Par exemple :

Code : Python

```
fichier = open("text.txt","r")
for ligne in fichier :
    print(ligne)
fichier.close()
```

Si vous exécuter ce code, vous pouvez d'ailleurs remarquer que le fichier de fin de ligne "\n" est aussi lu lors de la lecture du fichier ! Il ne faut pas l'oublier si vous voulez travailler sur ce qui a été lu.

Enfin, si vous ne voulez pas boucler directement sur le fichier, il est possible de récupérer l'ensemble de son contenu dans une

liste à l'aide de la fonction *readlines* (lire lignes) qui retourne l'ensemble des lignes du fichier sous forme de liste.

Code : Python

```
fichier = open("text.txt","r")
liste = fichier.readlines()
for element in liste :
    print(element)
fichier.close()
```

Ce code fait exactement la même chose que le précédent. Seulement, si vous vouliez travailler sur la liste des lignes avant, il est ici possible de le faire.

Se déplacer dans un fichier

Vous vous êtes peut-être posé la question de savoir comment le programme fait pour savoir où l'on en est dans le fichier lorsqu'on lui demande d'appliquer la fonction *readline*... Et bien la réponse est simple : la position actuelle dans le fichier est enregistrée ! Cette position est accessible via la fonction *tell* (dire) qui retourne la position actuelle du programme dans le fichier. Il est ensuite possible de déplacer le curseur de lecture du fichier à l'aide de la fonction *seek* (chercher). Et enfin, la fonction *readline* peut prendre en argument un nombre d'octets à lire. Cela vous permet donc de cibler parfaitement un endroit de votre fichier à lire. Par exemple, si l'on veut lire, dans notre fichier précédent, uniquement le mot "heureux", il suffit de faire :

Code : Python

```
fichier = open("text.txt","r")
#Je place mon curseur où il faut :
fichier.seek(15, 0)
#Je ne lis que le mot heureux :
mot = fichier.readline(7)
print(mot)
fichier.close()
```

La fonction *seek* accepte deux arguments. Le premier indique le nombre d'octet dont on souhaite se déplacer. Le second indique la position à partir de laquelle on souhaite se déplacer (0 pour commencer du début du fichier, 1 pour la position courante ou 2 pour la fin du fichier).

Vous avez maintenant une petite idée de ce qu'il est possible de faire avec des fichiers. Prenez le temps de vous entraîner à écrire et à lire des fichiers. Vous en aurez certainement besoin un jour !

Pour ceux que cela intéresse, il est aussi possible d'ouvrir les fichiers en mode binaire (suite de 0 et de 1). Cela se fait en rajoutant un 'b' dans le mode de lecture passé en paramètre de la fonction *open*. Ce mode de lecture et d'écriture de fichier n'est pas le plus utile ni le plus courant, c'est pourquoi je ne le présente pas ici, mais sachez que cela existe.

Vous savez maintenant ce qu'est une fonction. Si vous avez déjà en tête des idées de programmes qui vous intéressent, n'hésitez pas à consulter la documentation pour voir comment le faire ! Car tout est toujours faisable ! Il suffit juste de savoir quelle fonction permet de le faire... Maintenant, nous allons revenir sur les chaînes de caractères et voir quelques exemples de fonctions très utiles que l'on peut leur appliquer.

Les chaînes de caractères

Nous avons déjà travaillé avec des chaînes de caractères et pour cause, jusqu'à maintenant, nous nous contentons d'écrire des programmes qui fonctionnent en console. Vous allez voir que l'on peut faire plein de choses intéressantes sur les chaînes de caractères !

Dans cette partie, nous allons donc voir :

- Un rappel sur les chaînes de caractères
- Quelques fonctions bien pratiques
- Un exemple avec la création d'une syntaxe

Vous verrez que vous ne pourrez plus vous passez des chaînes de caractères après ce chapitre ! Bien les maîtriser vous permettra d'écrire et de lire des fichiers facilement par exemple, et ça, vous en aurez certainement besoin un jour ou l'autre ! Alors, au boulot !

Rappel

Nous avons utilisé des chaînes de caractères dans la plupart des chapitres précédents mais nous n'avons rien fait de très spécial jusqu'à maintenant ! Nous avons vu que ce sont des séquences, c'est-à-dire des suites ordonnées de caractères. Voyons en quelques essais dans le shell ce que nous avons vu jusqu'à maintenant :

Code : Python

```
>>> chaine = """Peuh !"""
>>> chaine[0]
'P'
>>> chaine * 3
'Peuh !Peuh !Peuh !'
>>> str(218)
'218'
```

Et c'est à peu près tout ce que l'on a vu jusqu'à maintenant... Effectivement, c'est le strict nécessaire pour réussir à écrire quelque chose en console ! Il existe d'autres petites opérations bien pratiques pour la manipulation des séquences en générales qui s'appliquent donc aux chaînes de caractères. La première est le découpage qui vous permet d'accéder à une sous partie de votre séquence et la deuxième est le test d'appartenance d'un élément à une séquence. Voyons ce que cela donne en shell :

Code : Python

```
>>> chaine = "Peuh ! Je suis une chaine !"
>>> chaine[3:]
'h ! Je suis une chaine !'
>>> chaine[:5]
'Peuh '
>>> chaine[3:5]
'h '
>>> "uh" in chaine
True
>>> "Je" not in chaine
False
```

Alors, le découpage se fait un peu de la même manière que l'accès à un élément d'une séquence par son indice. Ainsi, vous mettez entre crochets ([]) les indices auxquels vous voulez accédez et il vous retourne une chaîne de caractères tronquée comme vous le souhaitez. Remarquez que si aucun indice de départ ou de fin n'est spécifié, par défaut, ils seront définis au premier et au dernier indice de votre séquence.
Les opérateurs *in* (dans) et *not in* (pas dedans) vous permette de tester la présence d'un élément dans votre séquence. Ainsi, vous pouvez constater que 'uh' et 'Je' font effectivement partie de la séquence *chaine*. Ces opérateurs peuvent être utilisés pour créer des booléens dans la définition des structures conditionnelles ou dans les boucles *while* sans aucun problème. En général, leur lecture est simple car on compose presque une phrase en langage naturel (pour peu que l'on parle anglais...) !

Nous allons voir maintenant, que comme pour les fichiers, il est possible d'appliquer des fonctions aux chaînes de caractères.

Fonctions sur les chaînes

La première chose à faire lorsque l'on souhaite en connaître un peu plus sur ce que l'on peut faire avec certains objets, c'est de consulter la documentation ! Je vous donne donc déjà le lien de la documentation des types intégrées avec les fonctions que

vous pourrez appliquer à vos chaines de caractères (*str*) :

http://docs.python.org/py3k/library/stdtypes.html

Voyons maintenant ensemble quelques opérations plus ou moins utiles sur les chaînes de caractères. Pour cela, nous effectuerons quelques tests dans le shell. Nous allons commencer par les fonctions qui s'appliquent aux séquences en général et leurs effets sur les chaînes de caractères.

Code : Python

```
>>> chaine = 'Peuh !'
>>> len(chaine)
6
>>> min(chaine)
' '
>>> max(chaine)
'u'
>>> chaine2 = 'Pouet!'
>>> zip(chaine, chaine2)
<zip object at 0x01F9C120>
>>> for element in zip(chaine, chaine2) :
 print(element)

('P', 'P')
('e', 'o')
('u', 'u')
('h', 'e')
(' ', 't')
('!', '!')
```

Quelques explications sur ces fonctions s'imposent :

- **len** : diminutif de *length* (longueur), retourne la longueur de la séquence, c'est-à-dire le nombre d'éléments qui la composent.
- **min** : diminutif de minimum, retourne le plus petit élément de la séquence (dans notre cas, c'est l'ordre alphabétique qui est considéré)
- **max** : diminutif de maximum, retourne le plus grand élément de la séquence (considère aussi l'ordre alphabétique)
- **zip** : crée les couples d'éléments de même indice des séquences passées en argument. On remarque d'ailleurs ici que c'est un type d'objet *zip* qui est créé. Cet objet est itérable et j'utilise alors une boucle *for* pour en afficher le contenu.

Ces fonctions peuvent s'appliquer à toutes les séquences et ne sont pas spécifiques aux chaînes de caractères.

Nous allons tester des fonctions qui s'appliquent, elles, uniquement aux chaînes de caractères et dont on doit spécifier à quelle chaîne elle s'applique.

Code : Python

```
>>> chaine = "peuh !"
>>> chaine.capitalize()
'Peuh !'
>>> chaine.center(18)
' peuh ! '
>>> chaine.count("peu")
1
>>> chaine.find('uh')
2
>>> chaine.replace("p","f")
'feuh !'
>>> chaine.split()
['peuh', '!']
>>> chaine.split("h")
['peu', ' !']
>>> chaine.upper()
'PEUH !'
>>> chaine
```

```
'peuh !'
```

Il n'y a ici qu'un petit échantillon des fonctions que vous pouvez utilisez sur les chaînes de caractères. Si vous désirez une liste exhaustive, il faudra vous référer à la documentation ! Mais voyons tout de même ce que j'ai fait ici :

- **capitalize** ("majusculer") : retourne la même chaine mais avec une majuscule pour la première lettre.
- **center** (centrer) : retourne la chaîne entourée d'espaces et centrée dans un ensemble dont la taille est donnée en paramètre. Ici, ma chaîne fait 6 caractères et je demande de centrer sur 18, la fonction rajoute alors 6 espaces de chaque côté de ma chaîne.
- **count** (compter) : retourne le nombre d'occurrences de la chaîne passée en argument présentes dans la chaîne à laquelle la fonction est appliquée.
- **find** (trouver) : retourne l'indice de la première occurrence de la chaîne cherchée passée en argument.
- **replace** (remplacer) : retourne la chaîne de caractères avec toutes les occurrences de la première chaîne passée en argument remplacée par la deuxième chaîne passée en argument.
- **split** (séparer) : retourne une liste de chaînes de caractères correspondant aux différentes parties de la chaîne initiale coupée par la chaîne passée en argument. Par défaut, c'est le caractères ' ' (espace) qui est utilisé.
- **lower/upper** (plus petit/plus grand) : retourne la chaîne tout en minuscule/majuscule.

Ces fonctions ne modifient en rien la chaîne de départ. Elles se contentent de retourner un résultat qu'il faudra stocker dans une variable si vous désirez vous en servir plus tard ! J'ai d'ailleurs fait afficher la valeur de ma chaîne à la fin pour que vous en soyez convaincu !

Vous voyez donc que l'on peut faire de nombreuses opérations sur nos chaînes de caractères. Nous allons maintenant voir par un exemple, à quoi peuvent servir ces fonctions.

Exemple : créer une syntaxe

Maintenant que nous avons vu quelques opérations de bases sur la manipulation des chaînes de caractères, nous allons voir un exemple concret avec la création d'une syntaxe pour la lecture et l'écriture de fichiers (afin d'utiliser aussi un peu ce que vous avez appris dans les chapitres précédents ! ).

Tout d'abord, qu'appelle-t-on une syntaxe ? Et bien, c'est une sorte de mini langage structuré dont on va reconnaître certaines formes. Lorsque vous écrivez du code Python, vous utilisez une syntaxe spécifique qui est reconnue par l'interpréteur pour être transformée en code machine.

Quoi ?! Nous allons voir comment créer un langage de programmation ?! Mais je ne sais pas faire ça moi ! 

Ne vous inquiétez pas, nous allons nous contenter ici de créer une syntaxe très simple mais si vous voulez un jour créer votre propre langage de programmation, vous aurez déjà une petite idée de comment cela peut fonctionner !

Définition du problème

Commençons par définir notre problème et ce que l'on veut faire. Imaginons par exemple que l'on veuille à partir d'un fichier que l'on va écrire à la main, générer plein de fichiers remplis avec des tables de calculs. En gros, je veux donner des opérations à effectuer puis des nombres sur lesquels effectuer ces opérations. Cela pourrait donner un fichier d'entrée de la forme :

Code : Autre

```
+,-,*,/,%
12 38 42.3 98.345 375.32 23
[0,198,3]
```

Par exemple, avec ce fichier, je veux écrire pour chacune des opérations données à la première ligne et pour chacun des nombres de la deuxième, le résultat de l'opération du premier nombre considéré avec chacun des nombres de la troisième ligne. En clair, je veux la table d'addition de 12 avec les nombres de 0 à 198 par pas de 3, pareil avec les autres opérations et avec les autres nombres. Je veux aussi pouvoir mettre des commentaires dans mon fichier en utilisant un caractère spécifique. Pour ne pas être original, nous choisirons le caractère utilisé en Python : #.

L'utilisateur du programme doit pouvoir donner un nom de fichier à utiliser, ainsi qu'un nom de projet qui sera un repère pour les fichiers créés.

Ecrire un fichier de test

Cette partie est la plus simple ! Nous allons nous contenter d'écrire un fichier de test qui nous servira à lancer notre programme et voir si le résultat généré est correct. Vous devriez donc créer un fichier texte du nom que vous voulez (je l'appellerai *test.txt*) contenant :

Code : Autre

```
#Je suis un fichier de test :
+,-,*,/,%,tr                  #Je donne des opérations
12 38 42.3 98.345 375.32 23   #Puis des nombres
[0,198,3]                     #Puis d'autres nombres
```

L'opération 'tr' ne correspond à rien, cela va nous permettre de vérifier que les opérations qui n'existent pas ne seront effectivement pas prises en compte.

Ecrire le programme

Nous allons écrire le programme ensemble au fur et à mesure. Alors tout d'abord, je commence par un petit en-tête pour indiquer l'encodage de mon fichier mais aussi les quelques commentaires habituels tels que l'auteur du script et sa date de création :

Code : Python

```
# -*- coding: latin-1 -*-
##########################
# Création d'une syntaxe #
# ---------------------- #
# Auteur : Suzy #
# Crée le : 02/07/2010 #
##########################
```

Ensuite, l'utilisateur devra me donner un nom de fichier et j'espère bien que celui-ci existe. Il existe une fonction dans le module *os* de Python qui permet de vérifier l'existence d'un dossier ou d'un fichier. Je vais donc importer le module *os*.

Code : Python

```
# Utile pour vérifier l'existence d'un fichier :
import os
```

J'affiche ensuite une petite interface simple juste pour marquer le début de mon programme de façon claire dans la console :

Code : Python

```
# Petite interface textuelle :
print("*******************************\n" +
      "* Exemple de syntaxe simple : *\n" +
      "*******************************")
```

Afin de facilité la recherche des fichiers créés, nous allons demander un nom de projet à l'utilisateur :

Code : Python

```
# Demander le nom du projet :
nomProjet = input("Donnez un nom à votre projet : ")
```

Je veux maintenant récupérer le nom du fichier d'entrée que je devrais utiliser en vérifiant que celui-ci existe bien.

Code : Python

```
# Demander le nom du fichier à utiliser :
nomFichier = input("Quel fichier voulez-vous utiliser ? ")
while not os.path.exists(nomFichier) :
    print("Ce fichier n'existe pas !")
    nomFichier = input("Quel fichier voulez-vous utiliser ? ")
```

La condition de la boucle *while* est assez explicite si vous parlez un peu anglais. Elle dit "Tant que le chemin *nomFichier* n'existe pas fait :". Ceci est un excellent exemple de la simplicité du langage Python ! Le code est parfaitement lisible et compréhensible par n'importe qui !

La fonction *exists* du module *os.path* retourne *True* si le fichier/chemin existe et *False* sinon.

Maintenant que nous avons un nom de fichier correct, nous allons l'ouvrir et le lire ! Nous savons que seules trois lignes nous seront utiles. Nous allons donc compter les lignes afin d'identifier sur quelle ligne nous travaillons.

Code : Python

```
# Ouvrir le fichier :
fichier = open(nomFichier,"r")
numeroLigne = 1
# Lire le fichier :
for ligne in fichier :
    instructions...
```

Ici, on n'ouvre le fichier qu'en lecture car on ne veut pas le modifier. Maintenant va commencer l'analyse syntaxique. C'est-à-dire que nous allons traiter chacune des lignes du fichier pour en retirer les informations qui nous intéressent. Pour cela, on commence par nettoyer en jetant ce qui ne nous intéresse pas :

Code : Python

```
# Lire le fichier :
for ligne in fichier :
    # On retire le caractère de fin de ligne :
    ligne = ligne.replace("\n","")
    #On retire les commentaires de la lignes :
    ligne = ligne.split("#")[0]
    # On ne considère pas les lignes vides :
    if ligne != "" :
        instructions...
```

Le caractère de fin de ligne ne nous intéresse jamais, il vaut mieux le retirer tout de suite. Ensuite, les commentaires ne nous intéressent pas non plus. On retire donc tout ce qui se trouve après le caractère '#'. Enfin, si la ligne est vide après ces traitements, c'est qu'elle ne contient rien d'intéressant, sinon, c'est que c'est une des lignes qui nous intéressent. Une fois que l'on a potentiellement quelque chose qui nous intéresse, on va lire la ligne et surtout lire la syntaxe que nous avons créée :

Code : Python

```
# On ne considère pas les lignes vides :
    if ligne != "" :
        if numeroLigne == 1 :
            # On retire tous les espaces :
            ligne = ligne.replace(' ','')
            # On sépare les opérations :
            operations = ligne.split(',')
            print(operations)
```

```
            elif numeroLigne == 2 :
                # On sépare les nombres :
                premiersNombres = ligne.split(' ')
                # On retire tous les espaces de trop :
                while '' in premiersNombres :
                    premiersNombres.remove('')
                print(premiersNombres)
            elif numeroLigne == 3 :
                # On retire les crochets et les espaces et on sépare
les nombres :
                deuxiemesNombres = ligne.replace('
','').replace("[","").replace("]","").split(',')
                print(deuxiemesNombres)
            numeroLigne += 1
```

Dans l'exemple que je donne, une mauvaise écriture du fichier d'entrée entraînera des erreurs ! Nous ne gèrerons pas ici les erreurs de syntaxe dans le fichier d'entrée, mais vous serez parfaitement capable de le faire après le chapitre consacré aux exceptions.

Vous remarquerez ici que l'on peut utiliser les fonctions sur les chaînes de caractères soit séparément soit en les chaînant. En effet, le résultat de chacune de ces fonctions étant aussi une chaîne de caractères, on peut leur appliquer directement des fonctions ! Ce qui fait que dans l'analyse de la troisième ligne, le traitement ne tient qu'en une seule ligne !

Les *print* ne sont ici que pour vous écrire ce qui sort exactement de cette première partie du programme. Cela vous permet de vérifier notamment que l'analyse syntaxique est bonne !

Maintenant que l'on a récupéré les données qui nous intéressent dans notre fichier, il faut bien penser à le fermer :

Code : Python

```
#Fermer le fichier :
fichier.close()
```

Notre fichier d'entrée ne nous intéresse maintenant plus et nous allons nous attaquer à l'écriture des fichiers de sorties. On veut donc un fichier différent pour chacune des opérations et chacune des valeurs de la première liste. Ensuite, il faudra remplir les fichiers avec l'ensemble des opérations. On va donc avoir la chose suivante :

Code : Python

```
# Pour chaque opération :
for operation in operations :
    # Associer un nom à chaque opération et ne pas faire la suite
    # si l'opération n'existe pas :
    if operation == '+' :
        nomOperation = 'addition'
    elif operation == '-' :
        nomOperation = 'soustraction'
    elif operation == '*' :
        nomOperation = 'multiplication'
    elif operation == '/' :
        nomOperation = 'division'
    elif operation == '%' :
        nomOperation = 'modulo'
    elif operation == '**' :
        nomOperation = 'puissance'
    elif operation == '//' :
        nomOperation = 'divisionEntiere'
    else :
        print("L'opération %s n'est pas reconnue !" % (operation))
        continue
```

Ceci va permettre de ne considérer que les opérations qui existent effectivement ! L'utilisateur sera donc libre de taper n'importe quoi pour les opérations, il verra bien si cela ne fonctionne pas ! J'utilise ici pour la première fois l'outil de formatage des chaînes de caractères à la ligne :

Code : Python

```
print("L'opération %s n'est pas reconnue !" % (operation))
```

L'opérateur % permet de remplacer les éléments marqués par % par ceux données entre parenthèses par la suite. Le formatage des chaînes se fait via différents indicateurs, les principaux étant :

- **% s** : conversion en chaîne via *str*() avant formatage.
- **% i** : nombre entier (*integer*).
- **% e / % E** : notation exponentielle (scientifique) d'un nombre.
- **% f / % F** : nombre réel (à virgule flottante).
- **% %** : le caractère % en lui-même !

De cette façon, il est possible de formater facilement une chaîne de caractère en y insérant les valeurs de variables par exemple. C'est ce nous ferons dans la suite pour nommer correctement les fichiers que nous allons créer. Créons donc nos fichiers :

Code : Python

```
    # Pour chacune des valeurs :
        for valeur in premiersNombres :
            # Création du fichier :
            fichier = open('%s-%s-%s.txt' %
(nomProjet,nomOperation,valeur), 'w')
```

On va maintenant écrire les résultats de nos opérations dans les fichiers. Bien entendu, il faut faire attention à considérer les opérations impossibles à réaliser telles que la division ou le modulo par 0. On obtient alors :

Code : Python

```
# Ecrire dans le fichier :
        for nombre in
range(eval(deuxiemesNombres[0]),eval(deuxiemesNombres[1]),eval(deuxiemesNombres[2
:
            if (operation == '/' or operation == '//' or operation == '%') and
nombre == 0 :
                fichier.write("%s %s %s = impossible !\n" % (valeur.center(10),
operation, str(nombre).center(10)))
            else :
                fichier.write("%s %s %s = %f\n" % (valeur.center(10), operation,
str(nombre).center(10), \

eval(valeur+operation+str(nombre))))
```

Pour la première fois ici, j'écris une instruction Python sur plusieurs lignes. Pour cela, j'utilise le caractère '\' qui indique que l'instruction se poursuit à la ligne suivante. Du moment que vous utilisez ce caractère à chacune de vos lignes, il est possible d'écrire une instruction sur autant de lignes que vous le désirez.

Remarquez ici que j'utilise la spécificité de la fonction *eval* qui peut évaluer le résultat d'opérations écrites dans une chaîne de caractères ! J'évite ainsi plusieurs conversions inutiles de mes variables. Ensuite, j'utilise la fonction *center* pour aligner correctement mes résultats et obtenir des fichiers un peu plus lisibles. Enfin, il ne faut pas oublier de fermer le fichier que l'on vient d'ouvrir à chaque pas de boucle afin de s'assurer que celui-ci se sauvegarde bien.

Code : Python

```
    # Fermer le fichier :
```

```
    fichier.close()
```

En Python, vous ne pouvez pas ouvrir plus de 50 fichiers simultanément ! N'oubliez donc jamais de fermer les fichiers que vous ouvrez, surtout si vous en ouvrez dans une boucle comme c'est le cas ici ! Si vous ne le faites pas, votre programme a de fortes chances de planter !

Voilà, maintenant notre programme est terminé ! Je vous donne quand même la version complète pour que vous puissiez vous assurez de l'exécuter correctement ! Vous devriez obtenir tout un tas de fichiers textes très moches mais qui correspondent bien à ce que vous vouliez ! Vous avez donc réussi à analyser des lignes de texte et à en exploiter les données pour créer d'autres fichiers selon un format que vous avez créé !

Code : Python - v3

```
# -*- coding: latin-1 -*-
#########################
# Création d'une syntaxe #
# ---------------------- #
# Auteur : Suzy #
# Crée le : 02/07/2010 #
#########################

# Utile pour vérifier l'existence d'un fichier :
import os

# Petite interface textuelle :
print("*******************************\n" +
      "* Exemple de syntaxe simple : *\n" +
      "*******************************")

# Demander le nom du projet :
nomProjet = input("Donnez un nom à votre projet : ")

# Demander le nom du fichier à utiliser :
nomFichier = input("Quel fichier voulez-vous utiliser ? ")
while not os.path.exists(nomFichier) :
    print("Ce fichier n'existe pas !")
    nomFichier = input("Quel fichier voulez-vous utiliser ? ")

# Ouvrir le fichier :
fichier = open(nomFichier,"r")
numeroLigne = 1
# Lire le fichier :
for ligne in fichier :
    # On retire le caractère de fin de ligne :
    ligne = ligne.replace("\n","")
    #On retire les commentaires de la lignes :
    ligne = ligne.split("#")[0]
    # On ne considère pas les lignes vides :
    if ligne != "" :
        if numeroLigne == 1 :
            # On retire tous les espaces :
            ligne = ligne.replace(' ','')
            # On sépare les opérations :
            operations = ligne.split(',')
        elif numeroLigne == 2 :
            # On sépare les nombres :
            premiersNombres = ligne.split(' ')
            # On retire tous les espaces de trop :
            while '' in premiersNombres :
                premiersNombres.remove('')
        elif numeroLigne == 3 :
            # On retire les crochets et les espaces et on sépare les nombres :
            deuxiemesNombres = ligne.replace('
','').replace("[","").replace("]","").split(',')
        numeroLigne += 1
```

```
    #Fermer le fichier :
    fichier.close()

    # Pour chaque opération :
    for operation in operations :
        # Associer un nom à chaque opération et ne pas faire la suite
        # si l'opération n'existe pas :
        if operation == '+' :
            nomOperation = 'addition'
        elif operation == '-' :
            nomOperation = 'soustraction'
        elif operation == '*' :
            nomOperation = 'multiplication'
        elif operation == '/' :
            nomOperation = 'division'
        elif operation == '%' :
            nomOperation = 'modulo'
        elif operation == '**' :
            nomOperation = 'puissance'
        elif operation == '//' :
            nomOperation = 'divisionEntiere'
        else :
            print("L'opération %s n'est pas reconnue : les fichiers ne seront pas
    générés !" % (operation))
            continue
        print("Génération des fichiers pour l'opération %s." % (operation))
        # Pour chacune des valeurs :
        for valeur in premiersNombres :
            # Création du fichier :
            fichier = open('%s-%s-%s.txt' % (nomProjet,nomOperation,valeur), 'w')
            # Ecrire dans le fichier :
            for nombre in
    range(eval(deuxiemesNombres[0]),eval(deuxiemesNombres[1]),eval(deuxiemesNombres[
    :
                if (operation == '/' or operation == '//' or operation == '%') and
    nombre == 0 :
                    fichier.write("%s %s %s = impossible !\n" % (valeur.center(10),
    operation, str(nombre).center(10)))
                else :
                    fichier.write("%s %s %s = %f\n" % (valeur.center(10), operation,
    str(nombre).center(10), \

    eval(valeur+operation+str(nombre))))
            # Fermer le fichier :
            fichier.close()
```

Dans cette version, j'ai modifié légèrement l'affichage pour indiquer quelle opération est en cours de traitement.

Pour ceux qui travailleraient en Python v2.7, la seuls différence est encore une fois dans la fonction *print* qui ne nécessite pas de parenthèses et la fonction *input* qui doit être remplacée par la fonction *raw_input*.

Un bon entraînement serait de faire un programme qui fait exactement l'inverse ! En étudiant un ensemble de fichiers remplis de calculs, il pourrait générer un fichier d'entrée pour notre programme ci-dessus ! Enfin, si vous vous sentez le courage de le faire, le principe est le même !
Bien, vous maîtrisez maintenant un peu mieux les chaînes de caractères. N'hésitez surtout pas à vous exercer. Je suis sûr que vous trouverez de vous-mêmes des idées pour vous entraîner ! Si vous vous sentez prêt à continuer, nous allons étudier les listes dans le prochain chapitre.

Les listes

Nous avançons de plus en plus dans le langage Python. Nous avons vu beaucoup de choses mais nous ne nous sommes pas encore arrêtés sur les listes. Pourtant, vous les avez déjà utilisées jusqu'à maintenant sans trop savoir les manipuler. Nous allons donc voir dans cette partie :

- Quelques rappels sur les séquences et les listes
- Comment utiliser des listes
- Ce que l'on peut en faire

Après cette partie, les listes ne devraient plus avoir de secrets pour vous !

Rappels

Les listes sont des séquences ordonnées de données auxquelles on peut accéder à l'aide de leur indice associé. Faisons un rappel de ce que l'on a fait avec jusqu'à maintenant en effectuant quelques petits tests dans le shell :

Code : Python

```
>>> liste = ['Peuh !', 42, True, "Pouet !", "...", 90.2]
>>> liste[2]
True
>>> liste[-1]
90.2
>>> liste[2:]
[True, 'Pouet !', '...', 90.2]
>>> liste[:2]
['Peuh !', 42]
>>> liste[2:4]
[True, 'Pouet !']
>>> len(liste)
6
>>> for element in liste :
 print(element)


Peuh !
42
True
Pouet !
...
90.2
>>> for element in zip(liste, "123456") :
        print(element)

('Peuh !', '1')
(42, '2')
(True, '3')
('Pouet !', '4')
('...', '5')
(90.200000000000003, '6')
```

On a donc vu comment créer une liste et comment accéder à ses éléments à l'aide des indices ou d'une boucle *for*. Comme pour les chaînes de caractères, vous pouvez utiliser les fonctions intégrées qui s'appliquent à toutes les séquences :

- **len** : diminutif de *length* (longueur), retourne la longueur de la séquence, c'est-à-dire le nombre d'éléments qui la composent.
- **min** : diminutif de minimum, retourne le plus petit élément de la séquence (dans notre cas, c'est l'ordre alphabétique qui est considéré)
- **max** : diminutif de maximum, reoturne le plus grand élément de la séquence (considère aussi l'ordre alphabétique)
- **zip** : crée les couples d'éléments de même indice des séquences passées en argument. On remarque d'ailleurs ici que c'est un type d'objet *zip* qui est créé. Cet objet est itérable et j'utilise alors une boucle *for* pour en afficher le contenu.

Voilà ce que l'on sait des listes jusqu'à maintenant alors essayons d'aller plus loin.

Opérations de base

Créer une liste

Commençons par le commencement et voyons comment créer une liste ! Vous avez déjà vu que l'on peut créer une liste en la déclarant et en lui affectant directement une liste de données :

Code : Python

```
liste = ['Peuh !', 42, True, "Pouet !", "...", 90.2]
```

Mais à vrai dire, vous n'êtes pas obligé de lui attribuer tout de suite des données et il est parfaitement possible de déclarer une liste vide :

Code : Python

```
liste = []
```

Je ne pourrais bien entendu pas faire grand chose de cette liste vide hormis lui rajouter des éléments. Nous verrons comment faire juste après. Un autre moyen de créer une liste, est d'utiliser la fonction intégrée *list* qui permet de créer soit un liste vide si vous ne lui donnez aucun argument soit de créer une liste à partir d'un objet itérable que vous lui passez en argument :

Code : Python

```
>>> list()
[]
>>> list("Hahaha !")
['H', 'a', 'h', 'a', 'h', 'a', ' ', '!']
```

A l'aide de cette fonction, il est donc possible de transformer n'importe quel objet itérable en liste ! Il est aussi possible de créer une liste avec des conditions. Je m'explique. Par exemple, vous avez une première liste de nombres. Vous pouvez en créer une deuxième en une seule ligne en vous basant sur une condition :

Code : Python

```
>>> liste = list(range(100))
>>> sousliste = [nombre for nombre in liste if nombre < 20]
>>> sousliste
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
19]
```

Ici, je crée une première liste de nombres de 0 à 99. Je déclare ensuite une variable *sousliste* que je veux remplir avec tous les éléments de *liste* inférieurs à 20. J'écris donc une phrase logique "Pour tout nombre dans liste inférieur à 20". Cette forme n'est pas des plus simples à maîtriser mais vous vous y ferez. et si vous ne voulez pas l'utiliser, vous pouvez toujours le faire en quelques lignes avec une boucle *for* qui ne soit pas intégrée directement dans la déclaration de la liste !

Les tuples

En Python, il existe un autre type de liste : les tuples. Les tuples se manipulent exactement comme les listes, à une différence près, leurs éléments ne peuvent être modifiés. Si vous avez donc une liste de constantes à définir, il vaudra mieux utiliser un tuple pour ne pas prendre le risque de les modifier au cours de votre programme. Il sera toujours possible au cours de votre programme d'ajouter des données dans un tuple à l'aide d'une addition de tuple par exemple.

Bien que 'tuple' soit un mot anglais prononcé 'teupeule', c'est aussi un mot bien français de la même famille que quintuple, sextuple, etc... La seule différence est qu'il désigne un ensemble d'objets sans en préciser la quantité. On ne pourra donc pas vous reprocher d'utiliser un anglicisme !

Les tuples se déclarent comme les listes mais sont délimités par des parenthèses '()' et non pas par des crochets '[]'. Ainsi, pour déclarer un tuple, il suffit de faire :

Code : Python

```
>> unTuple = tuple("Peuh !")
>>> unTuple
('P', 'e', 'u', 'h', ' ', '!')
>>> unTuple = ('Peuh !', 23, 34, 45, True)
>>> unTuple
('Peuh !', 23, 34, 45, True)>
```

Vous voyez donc qu'il existe une fonction *tuple* qui s'utilise de façon équivalente à la fonction *list*. Vous pouvez accéder aux données contenues dans un tuple exactement de la même manière que pour une liste en utilisant les indices ou les fonctions associées.

Maintenant que vous savez créer des listes, nous allons voir comment les modifier.

Modifier une liste

Il existe bien des manières de modifier une liste... Vous pouvez la modifier dans son intégralité en lui affectant une nouvelle valeur, par exemple :

Code : Python

```
>>> liste = [23,32,43,45,56]
>>> liste = liste * 2
>>> liste
[23, 32, 43, 45, 56, 23, 32, 43, 45, 56]
```

OU

Code : Python

```
>>> liste = [23,32,43,45,56]
>>> liste
[23, 32, 43, 45, 56]
>>> liste = ["Peuh !", "Pouet !"]
>>> liste
['Peuh !', 'Pouet !']
```

Ici, j'ai créé une liste de nombres et j'ai décidé de la modifier en répétant deux fois la liste donnée initialement dans le premier cas et en lui affectant carrément une nouvelle liste dans le deuxième cas. Vous voyez qu'après cette opération, ma liste a effectivement été modifiée. Mais il est aussi possible de ne modifier qu'un seul élément dans la liste :

Code : Python

```
>>> liste
[23, 32, 43, 45, 56, 23, 32, 43, 45, 56]
>>> liste[2] = 200
>>> liste
[23, 32, 200, 45, 56, 23, 32, 43, 45, 56]
```

Si vous souhaitez modifier tous les éléments d'une liste un par un à l'aide d'une boucle *for*, vous devrez utiliser une fonction particulière. Vous allez tout de suite comprendre pourquoi, regardez cet exemple :

Code : Python

```
>>> liste
[23, 32, 43, 45, 56, 23, 32, 43, 45, 56]
>>> for element in liste :
 element += 100
```

```
>>> liste
[23, 32, 43, 45, 56, 23, 32, 43, 45, 56]
```

Vous pouvez constater qu'aucun élément n'a été modifié dans la liste et c'est normal ! Car ici, c'est la variable *element* qui a été modifiée et elle ne fait pas partie intégrante de la liste ! Pour pouvoir modifier la liste élément par élément, il vous faudra donc accéder aux indices au sein de la liste. Pour ça, vous avez plusieurs solutions :

Code : Python

```
>>> liste
[23, 32, 200, 45, 56, 23, 32, 43, 45, 56]
>>> for i in range(len(liste)) :
 liste[i] += 100
>>> liste
[123, 132, 300, 145, 156, 123, 132, 143, 145, 156]
```

OU

Code : Python

```
>>> liste
[23, 32, 43, 45, 56]
>>> for i, element in enumerate(liste) :
 liste[i] += 100
>>> liste
[123, 132, 143, 145, 156]
```

Dans le premier cas, vous utilisez les fonctions intégrées *len* pour obtenir la taille de la liste et la fonction *range* pour générer une liste des indices. Dans le second cas, la fonction intégrée *enumerate* (énumérer) vous retourne la liste des couples (indice, élément) de l'ensemble de votre liste. Ainsi, de cette manière, vous avez accès à la fois aux éléments de votre liste mais aussi à leur indice ! Nous avons vu ici comment modifier des éléments déjà présents dans une liste, mais il est possible d'ajouter et de retirer des éléments d'une liste ! Des fonctions appliquées aux listes permettent d'effectuer ce genre d'action :

Code : Python

```
>>> liste = [23, 34, 45, 56, 67]
>>> liste.append(78)
>>> liste
[23, 34, 45, 56, 67, 78]
>>> liste.insert(2, 109)
>>> liste
[23, 34, 109, 45, 56, 67, 78]
>>> liste.sort()
>>> liste
[23, 34, 45, 56, 67, 78, 109]
>>> liste.sort(reverse = True)
>>> liste
[109, 78, 67, 56, 45, 34, 23]
>>> liste.reverse()
>>> liste
[23, 34, 45, 56, 67, 78, 109]
>>> liste.remove(78)
>>> liste
[23, 34, 45, 56, 67, 109]
>>> del liste
>>> liste
Traceback (most recent call last):
  File "<pyshell#104>", line 1, in <module>
    liste
NameError: name 'liste' is not defined
```

J'applique ici des fonctions directement à ma liste. Remarquez que ces dernières la modifient directement. Ces fonctions sont donc :

- **append** (ajouter) : ajoute l'élément passé en argument à la fin de la liste.
- **insert** (insérer) : insère dans la liste à l'indice donné en premier argument l'élément donné en deuxième argument.
- **sort** (trier) : range dans l'ordre croissant les éléments de la liste (l'ordre peut être décroissant si l'argument *reverse* est mis à *True*).
- **reverse** (inverser) : inverse l'ordre des éléments dans la liste.
- **remove** (retirer) : retire l'objet passé en argument de la liste. S'il y est présent plusieurs fois, seule la première occurrence sera retirée.

Le mot clé *del* (diminutif de *delete* (supprimer)) suivi du nom de la liste la supprime carrément des variables déclarées ! Elle n'existe donc plus en mémoire après effacement avec *del*. C'est d'ailleurs pour cela qu'une erreur se produit puisque ma variable *liste* n'existe plus, je ne peux donc plus afficher sa valeur...

Pour une liste exhaustive sur les fonctions applicables aux tuples et aux listes, je vous laisse consulter la documentation qui reste le document le plus complet que vous puissiez vous procurez !

Nous allons voir maintenant comment manipuler les listes et ce que l'on peut en faire.

Manipuler les listes

Maintenant que l'on sait créer et modifier le contenu des listes, nous allons revoir comment les parcourir et rapidement à quoi cela peut servir. Voyons d'abord ce que l'on peut faire avec les fonctions intégrées :

Code : Python

```
>>> liste = [23,45,56,34]
>>> sorted(liste)
[23, 34, 45, 56]
>>> reversed(liste)
<list_reverseiterator object at 0x02064EB0>
>>> for element in reversed(liste) :
 print(element)
34
56
45
23
>>> sum(liste)
158
>>> len(liste)
4
>>> min(liste)
23
>>> max(liste)
56
>>> liste
[23, 45, 56, 34]
```

J'utilise donc différentes fonctions intégrées qui ne modifient pas ma liste de départ :

- **sorted** (rangé) : retourne la même liste mais avec les éléments rangés dans l'ordre croissant.
- **reversed** (inversé) : retourne la même liste mais avec les éléments dans l'ordre inverse.
- **sum** (somme) : retourne la somme de tous les éléments de la liste.
- **len** (longueur) : retourne le nombre d'éléments dans la liste.
- **min/max** (minimum/maximum) : retourne l'élément le plus petit/grand de la liste.

Ces fonctions présentent l'avantage de ne pas modifier nos listes et n'altèrent donc pas les données stockées.

Il est possible d'imbriquer des listes pour générer des tableaux ou matrices à plusieurs dimensions. Prenons un exemple :

Code : Python

```
>>> liste = []
>>> for i in range(10) :
 liste.append(list(range(10)))
```

```
>>> for element in liste :
 print(element)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

On voit donc ici que j'ai créé une liste de liste. Pour n'accéder qu'à un seul élément dans une de ces listes, il suffit alors de spécifier les indices de cette manière :

Code : Python

```
>>> liste[2][3]
3
```

Le deuxième élément de *liste* étant lui-même une liste, il est alors possible d'accéder à ses éléments à l'aide de leur indice et c'est ce que je fais ici. Pour parcourir une liste de liste, on peut alors imbriquer les boucles *for* :

Code : Python

```
for ligne in liste :
    for element in ligne :
        print(element)
```

Ce code vous affiche tous les éléments présents dans toutes les listes contenues dans la liste principale !

Les listes sont assez simples à manipuler et vous en utiliserez assez souvent... Leur maîtrise vient avec l'expérience et plus vous les utiliserez, plus leur utilisation sera instinctive.

Toutes les fonctions qui permettent de manipuler des listes sont aussi valables pour les tuples. Seules les fonctions altérant vos listes ne sont pas applicables aux tuples puisque leurs éléments ne peuvent être modifiés. N'hésitez pas à consulter la documentation à ce sujet !

Bien, maintenant que vous en savez un peu plus sur ce qu'il y a à savoir sur les listes, passons à un autre outil de stockage de données bien pratique : les dictionnaires.

Les dictionnaires

Après avoir étudié les séquences avec les chaînes, les listes et les tuples, nous allons nous intéresser à un autre type de données : les types associatifs. En Python, il n'existe qu'un seul type associatif appelé dictionnaire. Vous comprendrez assez vite pourquoi celui-ci s'appelle ainsi.

Nous allons donc voir dans cette partie :

- Ce qu'est un type associatif
- Comment créer et manipuler des dictionnaires
- A quoi cela peut servir

Ce sont des objets assez simples à utiliser mais ils sont souvent délaissés car on y pense trop peu souvent... Alors essayez d'en comprendre l'utilité !

Qu'est-ce que c'est ?

De quoi va-t-on parler exactement dans ce chapitre ? Et bien, nous allons apprendre à utiliser un type de donnée associatif : les dictionnaires. Le dictionnaire en Python permet de stocker un ensemble données au même titre qu'une liste ou un tuple. Seulement, les données n'y seront pas ordonnées...

Mais alors j'y accède comment aux données dans ce dictionnaire ?!

C'est là qu'intervient toute la magie de l'associatif : vous pourrez accéder aux données à l'aide d'une clé (ou valeur de hachage). En fait, dans un dictionnaire, chaque fois que vous voudrez ajouter une donnée, vous devrez l'associer à une clé pour former le couple (clé, valeur). C'est cette clé qui vous permettra d'accéder aux données stockées au même titre que l'indice le permet dans les séquences.

Pour une première approche, nous allons faire quelques essais dans le shell en créant un dictionnaire et en essayant d'accéder à ses éléments :

Code : Python

```
>>> dico = {"Peuh !" : "Bonjour !", "J'te déteste !" : "Je
t'apprécie !", "J'espère ne jamais te revoir !" : "A bientot !"}
>>> dico["Peuh !"]
'Bonjour !'
>>> dico["J'te déteste !"]
"Je t'apprécie !"
```

Dans cet exemple, je crée un dictionnaire contenant trois couples (clé, valeur). Le dictionnaire est marqué par les accolades '{}', chaque clé est associée à sa valeur par ':' et les couples sont séparés par des virgules. Afin de rendre l'exemple un peu plus parlant, je l'utilise comme un vrai dictionnaire. J'associe ici à chacune de mes phrases favorites sa réelle signification. Et vous pouvez voir que pour accéder à un élément, j'utilise bien les crochets '[]' comme pour accéder à un élément d'une séquence, mais ici l'indice est remplacé par une clé.

L'important à savoir, c'est que n'importe quel type de donnée peut-être utilisé comme clé d'un dictionnaire ! Seulement, si cette donnée est modifiée, cela peut engendrer des erreurs. C'est pourquoi il vaut mieux utiliser des données dites immuables comme clé. Les données immuables sont des nombres, des chaînes de caractères et les tuples. Evitez donc d'utiliser des variables comme clé car si leur valeur change, alors elle ne correspondent plus à la clé définie initialement et ça risque de devenir très vite n'importe quoi ! Regardez :

Code : Python

```
>>> dico = {variable : "Hey !", variable2 : "Salut !", variable3 :
"Peuh !"}
>>> dico[variable]
'Hey !'
>>> variable = 2
>>> dico[variable]
'Salut !'
```

J'ai utilisé *variable* pour définir une clé, mais la clé n'est pas associé à la variable elle-même mais à sa valeur au moment de la déclaration du dictionnaire... En utilisant une variable comme clé, vous utilisez en réalité sa valeur actuelle comme clé et non pas la référence que représente la variable elle-même...

Opérations de base

Nous allons voir maintenant comment utiliser les dictionnaires et commençons par leur création.

Créer un dictionnaire

Nous avons vu juste avant comment déclarer un dictionnaire en lui affectant directement des couples (clé, valeur) :

Code : Python

```
dico = {clé1 : valeur1, clé2 : valeur2, ...}
```

Vous pouvez très bien déclarer un dictionnaire vide et lui rajouter des éléments par la suite. Cela se fait un peu comme pour les listes vides :

Code : Python

```
dico = {}
```

OU

Code : Python

```
dico = dict()
```

La fonction intégrée *dict* s'utilise un peu de la même manière que *list*. Vous pouvez l'utiliser sans argument pour créer un dictionnaire vide comme dans l'exemple ci-dessus ou en spécifiant un ou plusieurs arguments. Nous allons voir ce qu'il est possible de faire :

Code : Python

```
>>> dict(zip(('clé1', 'clé2', 'clé3'), (1, 2, 3)))
{'clé2': 2, 'clé3': 3, 'clé1': 1}
>>> dict([["clé1", 1], ["clé2", 3], ["clé3", 5]])
{'clé2': 3, 'clé3': 5, 'clé1': 1}
>>> dict([('abcdef'[i], i) for i in range(6)])
{'a': 0, 'c': 2, 'b': 1, 'e': 4, 'd': 3, 'f': 5}
>>> dict(a=1, b=2, c=3, d=4)
{'a': 1, 'c': 3, 'b': 2, 'd': 4}
```

Il existe donc bien des manières de créer un dictionnaire ! Etudions les un peu plus en détail :

- **1er cas** : nous avons vu plus tôt dans le cours que la fonction *zip* crée une liste de couples correspondant aux éléments de deux séquences passées en arguments. Ici, on passe donc l'équivalent d'une liste de couple en argument à la fonction *dict*().
- **2ème cas** : on passe ici une liste de listes de taille 2 qui correspondent aux couples (clé, valeur).
- **3ème cas** : on crée une liste de tuples de tailles 2 à l'aide d'une boucle *for*. Ces tuples correspondent aux couples (clé, valeur) du dictionnaire.
- **4ème cas** : on associe des valeurs à des clés directement à l'aide du signe =. Remarquez que les clés sont automatiquement converties en chaînes de caractères afin de les rendre immuables.

Vous savez maintenant comment créer un dictionnaire. Nous allons donc voir maintenant comment les modifier.

Modifier un dictionnaire

Pour rajouter une donnée dans un dictionnaire, il n'y a pas besoin de faire appel à une fonction comme pour les listes. Il suffit de déclarer un couple (clé, valeur) de la façon suivante :

Code : Python

```
>>> dico = {}
>>> dico['clé'] = 18
>>> dico
{'clé': 18}
```

Vous pouvez constater que la clé 'clé' n'était pas associé à la moindre valeur et pourtant je peux modifier la valeur qui y est associée ! Cela rajoute donc automatiquement le couple ('clé', 18) dans mon dictionnaire. Si vous travaillez avec plusieurs dictionnaires, vous pouvez aussi vous servir de l'un pour remplir l'autre à l'aide de la fonction *update* :

Code : Python

```
>>> dico1 = dict([('abcdef'[i], i) for i in range(6)])
>>> dico2 = dict(zip(('clé1', 'clé2', 'clé3'), (1, 2, 3)))
>>> dico1.update(dico2)
>>> dico1
{'a': 0, 'c': 2, 'b': 1, 'e': 4, 'd': 3, 'f': 5, 'clé1': 1, 'clé2':
2, 'clé3': 3}
```

Cette fonction rajoute au dictionnaire auquel elle est appliquée tous les couples (clé, valeur) du dictionnaire passé en argument. Il existe aussi des fonctions qui permettent d'effacer vos dictionnaires :

Code : Python

```
>>> dico1.clear()
>>> dico1
{}
>>> del dico1
>>> dico1
Traceback (most recent call last):
  File "<pyshell#31>", line 1, in <module>
    dico1
NameError: name 'dico1' is not defined
```

La fonction *clear* efface tous les couples (clé, valeur) du dictionnaire auquel elle est appliquée tandis que le mot clé *del* désalloue la mémoire du dictionnaire et le retire donc des variables déclarées ! Bien sûr, vous n'êtes pas obligé d'être aussi radical et vous pouvez décider de ne retirer qu'une seule valeur de votre dictionnaire. Pour cela, vous allez encore utiliser le mot clé *del* mais en spécifiant un élément en particulier dans le dictionnaire :

Code : Python

```
>>> dico1 = dict([('abcdef'[i], i) for i in range(6)])
>>> del dico1['a']
>>> dico1
{'c': 2, 'b': 1, 'e': 4, 'd': 3, 'f': 5}
```

Récupérer des données

La partie la plus intéressante dans les dictionnaires est de récupérer les données qui y sont contenues ! En général, les dictionnaires ne sont pas beaucoup modifiés après leur création et resteront assez constants. Toute leur utilité viendra alors de notre capacité à en faire quelque chose. Nous avons déjà vu qu'il est possible d'accéder à une valeur dans un dictionnaire de la même manière que lors de l'accès direct à un élément dans une séquence en remplaçant l'indice par une clé.

Mais qu'en est-il si l'on veut pouvoir boucler entièrement sur un dictionnaire par exemple ? Et bien, ce n'est pas si compliqué !

Des fonctions vont vous permettre de faire ça :

Code : Python

```
>>> dico1 = dict([('abcdef'[i], i) for i in range(6)])
>>> for cle in dico1 :
 print(cle)
a
c
b
e
d
f
```

OU

Code : Python

```
>>> dico1 = dict([('abcdef'[i], i) for i in range(6)])
>>> for cle, valeur in dico1.items() :
 print(cle, valeur)
a 0
c 2
b 1
e 4
d 3
f 5
```

La fonction *items* vous permet de récupérer les couples alors que si vous itérez directement sur le dictionnaire, vous ne récupérez que les clés.

De même, si vous voulez savoir si une clé se trouve dans un dictionnaire, vous pouvez utiliser les opérateurs *in* et *not in* :

Code : Python

```
>>> dico1 = dict([('abcdef'[i], i) for i in range(6)])
>>> 'a' in dico1
True
>>> 't' in dico1
False
>>> 't' not in dico1
True
>>> 'b' not in dico1
False
```

Vous avez donc un opérateur qui vous permet de tester facilement si une clé est présente ou non dans votre dictionnaire.

Si vous désirez une liste exhaustive des fonctions applicables aux dictionnaires, la documentation est comme toujours un excellent endroit où chercher !

A quoi ça sert ?

Après vous avoir présenté comment créer et modifier un dictionnaire, je vous propose d'en voir une petite application. Celle-ci peut paraître un peu sale et inutile mais je suis sûr que vous saurez l'apprécier en allant plus loin dans le cours ! Dès le chapitre suivant, je suis sûr que vous en verrez l'utilité !

Nous allons utiliser notre dictionnaire pour contrôler le flux d'instructions ! Et oui ! Vous allez voir que l'on peut remplacer une suite de *if/elif/else* en utilisant simplement un dictionnaire. Prenons par exemple un petit programme qui demande une information à l'utilisateur et qui exécute une fonction différente en fonction de l'entrée de l'utilisateur :

Code : Python

```
nombre = input("Entrez un nombre : ")

if nombre == '0' :
    fonction0()
elif nombre == '1' :
    fonction1()
elif nombre == '2' :
    fonction2()
...
...
...
```

C'est particulièrement lourd à écrire et ce n'est pas terrible à lire pour celui qui regardera votre code... Et bien, sachez que les dictionnaires ne sont pas obligé de contenir des données de types alphanumériques, des listes, des tuples ou tout ça. Les dictionnaires peuvent aussi contenir des noms de fonction ! De ce fait, on peut remplacer le code précédent par :

Code : Python

```
nombre = input("Entrez un nombre : ")

dico = {'0':fonction0,'1':fonction1, '2':fonction2,...}
dico[nombre]()
```

En faisant cela, l'interpréteur Python transformera la ligne *dico[nombre]* par le nom de la fonction stocké dans le dictionnaire. Et pour appeler une fonction, on écrit son nom que l'on fait suivre de parenthèses. Cela fera donc appel à la fonction que vous vouliez ! C'est un peu ça la magie du Python ! 

Vous avez presque fini d'acquérir les bases de Python ! Vous serez bientôt capable de devenir un programmeur aguerri capable de développer de vrais projets professionnels. En attendant, je vous propose de continuer sur les fonctions en apprenant à les écrire vous-mêmes !

Les fonctions (1/2)

Bien, dans cette partie, nous allons voir comment créer nos propres fonctions ! Cela passera par :

- Les différentes parties d'une fonction
- Comment ça marche
- Comment bien écrire une fonction
- Comment fonctionne l'espace mémoire
- Comment utiliser les valeurs par défaut

Il y a pas mal de choses à voir sur les fonctions et vous ne verrez que les bases dans cette partie. Vous en verrez un peu plus dans la partie de cours qui suit.

Ecrire des fonctions

Nous avons utilisé pas mal de fonctions jusqu'à maintenant pour effectuer tout un tas d'opérations sur les chaînes de caractères, les listes, les dictionnaires, afficher du texte dans la console, etc... Mais vous ne savez toujours pas en écrire par vous-même ! Commençons par quelques petits rappels sur ce qui définit une fonction.

Rappel

Nous avons vu qu'une fonction est composée de trois grandes parties :

- **Son nom** qui permet d'y faire appel.
- **Ses arguments** qui permet de spécifier des données à lui transmettre.
- **Sa sortie**, c'est-à-dire ce qu'elle retourne comme résultat.

son nom et ses arguments forment ce que l'on appelle la signature d'une fonction. Deux fonctions seront différenciées par l'interpréteur à partir du moment où elles ne possèdent pas la même signature. Faites donc attention à ne pas appeler vos fonctions de façon identique si elles possèdent les mêmes arguments, l'interpréteur ne pourra pas les différencier ! Comme nous n'en sommes qu'à la définition d'un peu de vocabulaire, il existe deux types de fonctions :

- **Les fonctions** qui effectuent des instructions et retournent un résultat.
- **Les procédures** qui effectuent des instructions mais ne retournent rien du tout.

Dans la pratique, les fonctions et les procédures s'appellent exactement de la même manière ! La seule différence réside effectivement dans le fait que les procédures ne renvoient pas de résultats et ne servent donc qu'à effectuer une suite d'instructions. Ensuite, nous avons vu que pour faire appel à une fonction, il faut écrire quelque chose du genre :

Code : Python

```
variable = fonction(argument1, argument2, ...)
```

OU

Code : Python

```
variable = objet.fonction(argument1, argument2, ...)
```

Il est donc possible d'appeler des fonctions, comme les fonctions intégrées, par exemple *print* ou *input*, ou d'appliquer des fonctions sur des objets comme les séquences, les fichiers ou les dictionnaires. Bien sûr, dans la pratique, vous n'êtes pas obligé de stocker la valeur retournée dans une variable. Pour les procédures, cela ne présente même aucun intérêt. Cependant, lorsque vous appelez une fonction, le plus souvent, c'est pour en obtenir le résultat !

Définition d'une fonction

Pour définir une fonction, vous avez besoin du mot clé *def*. La définition de fonction fonctionne alors comme l'écriture d'une structure. Définissons notre première fonction. Pour plus de commodité, nous ne travaillerons plus dans le shell mais dans des fichiers. Si vous n'y êtes pas encore habitué, revoyez les premiers chapitres !

Code : Python

```
def maPremiereProcedure() :
    print("J'ai fait appel à ma première procédure !")
    print("Je suis vraiment content !")
```

Nous avons défini ici une première procédure mais il serait intéressant de voir comment y faire appel. Et bien comme une fonction habituelle :

Code : Python

```
maPremiereProcedure()
```

Si vous insérez ces deux morceaux de code dans un fichier .py et que vous l'exécutez, vous devriez voir la magie opérer ! Vous remarquerez d'ailleurs que le code au sein des fonctions n'est exécuté que si l'on y fait appel. Vous pouvez donc y écrire ce que vous voulez sans que cela ne soit exécuter avant d'appeler explicitement les fonctions !

Les fonctions et les procédures doivent toujours être définies avant d'y faire appel comme pour les variables ! Autrement, l'interpréteur vous fera remarquer que la fonction que vous essayez d'appeler n'est pas définie.

Vous pouvez maintenant créer autant de procédures que vous voulez ! Voyons comment y ajouter des paramètres :

Code : Python

```
def maDeuxiemeProcedure(aEcrire, peuh) :
    print(aEcrire)
    print(peuh)

maDeuxiemeProcedure("HAHAHA !", 34)
```

La deuxième procédure est maintenant capable de prendre en compte deux arguments. J'ai donc maintenant deux variables que je peux utiliser dans ma procédure. Voyons maintenant comment créer une fonction qui retournera un résultat. La déclaration est identique mais pour retourner un résultat, vous devrez utiliser le mot clé *return* (retourner) suivi de ce que vous voulez retourner.

Code : Python

```
def puissance(nombre, puissance) :
    print("Je calcule %f à la puissance %f" % (nombre, puissance))
    return nombre**puissance

print(puissance(3,12))
```

La fonction puissance affiche alors ce qu'elle va calculer et retourne le résultat escompté.

Le mot clé *return* marque la fin de la fonction et aucune instruction ne sera exécutée dans la fonction après avoir retourné quelque chose.

Il est possible de retourner n'importe quel type de résultat ! Vous pouvez donc retourner des listes, des dictionnaires, des nombres, des chaînes de caractères et ce qu'il vous importe ! Une des grandes spécificités de Python est qu'il est possible de retourner plusieurs résultats ! L'ensemble des résultats sera alors mis sous forme de tuple immuable. Vous pouvez donc faire quelque chose de ce type :

Code : Python

```
def operations(nombre, facteur) :
    print("J'aime les opérations !")
    return nombre+facteur, nombre-facteur, nombre/facteur,
nombre*facteur

print(operations(3,12))
```

Vous avez sûrement remarqué jusqu'à maintenant qu'aucun type n'est précisé pour les arguments des fonctions... Le typage dynamique autorise donc n'importe quel type de donnée comme argument de vos fonctions. Seulement, cela posera certainement problème si les arguments ne sont pas du type attendu. C'est pourquoi, il faut absolument bien documenter ses fonctions afin de s'assurer de leur bonne utilisation ! Nous allons donc voir tout de suite comment documenter nos fonctions.

Documenter une fonction

Une dernière chose très importante lorsque vous définissez une fonction, c'est sa documentation ! Vous allez commencer à documenter votre code pour le rendre plus compréhensible et exploitable par d'autres personnes que vous. Pour cela, c'est très simple, il suffit de placer immédiatement sous la ligne de définition de votre fonction, une chaîne de caractères entre triple 'quote' (""" ou ''') qui sera considérée un commentaire.

Code : Python

```
def fonction(arg1, arg2) :
    """
Je documente ma fonction. J'explique rapidement à quoi sert cette
fonction.
arg1 : J'écris à quoi correspond cet argument.
arg2 : J'écris à quoi correspond cet argument.
return : J'indique ce que retourne la fonction
"""
    pass
```

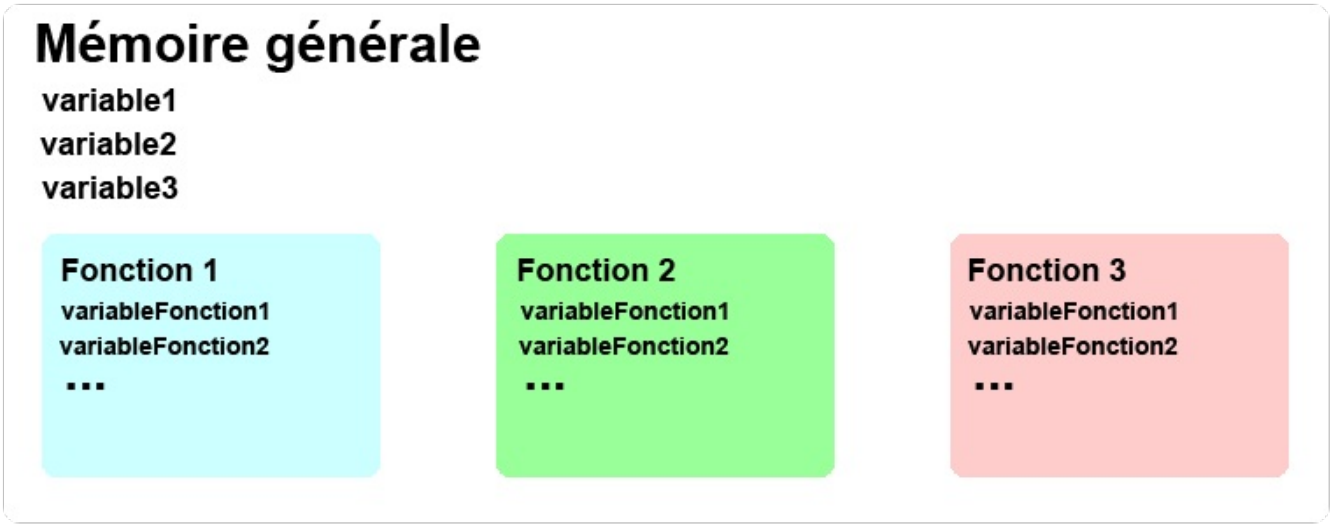
Voici un exemple d'une manière simple de documenter clairement vos fonctions. Il faut au minimum expliquer ce que fait la fonction. Une documentation complète détaille aussi le sens de chacun des arguments et ce que retourne la fonction. Avec une bonne documentation, personne ne doit avoir besoin de lire le code pour comprendre à quoi sert la fonction et ce qu'elle fait. Cela permet de la conserver en tant que boîte noire qui ne regarde que celui qui l'a implémentée.

Le mot clé *pass* vous permet de sortir d'une structure. Il peut être utile par exemple lorsque vous développez un projet. Une bonne technique de développement étant de définir les fonctions puis l'algorithme principal et enfin d'implémenter les fonctions. Ce mot clé vous permet donc d'avoir toutes vos fonctions définies sans avoir à les implémenter à l'avance.

Nous avons donc vu comment déclarer et appeler une fonction mais maintenant, nous allons voir comment est géré l'espace mémoire avec les fonctions.

Espace mémoire

L'espace mémoire en Python est géré de façon particulière. Les variables que vous définissez dans vos fonctions ne sont définies qu'au sein de ses fonctions par défaut. Tout ce qui est défini à l'extérieur de vos fonctions est défini dans l'espace mémoire général du programme. On a donc le schéma suivant :



Dans ce schéma, vous voyez des variables définies dans l'espace général et des variables définies dans les fonctions. Les

variables que vous utilisez au sein des fonctions peuvent d'ailleurs toutes avoir le même nom d'une fonction à l'autre, cela n'engendrera absolument aucune erreur et les fonctions n'interagiront pas les unes avec les autres par inadvertance !

Vous pouvez faire le test :

Code : Python

```
def fonction1() :
    """Fonction de test 1"""
    variable = 10
    variable **= 2
    print("Variable = %f" % variable)

def fonction2() :
    """Fonction qui ne devrait pas pouvoir fonctionner !"""
    print("Variable = %f" % variable)

fonction1()
fonction2()
```

Cela doit vous donner une belle erreur qui vous indique que *variable* n'est pas défini.

Code : Console

```
Variable = 100.000000
Traceback (most recent call last):
  File "G:/Travail/Site du zÃ©ro/Langage Python/Exemples/Memoire.py", line 10, in <
  File "G:/Travail/Site du zÃ©ro/Langage Python/Exemples/Memoire.py", line 7, in fc
NameError: global name 'variable' is not defined
```

Vous voyez d'ailleurs qu'il est indiqué que l'erreur s'est produite à la ligne 10, puis à la ligne 7. Cela vous permet notamment de cibler l'erreur de façon plus précise ! Avec cette erreur, vous pouvez donc constater par vous-même que chaque fonction possède son propre espace mémoire et ne peux pas accéder directement à l'espace mémoire général. Pour permettre cela, il existe un mot clé : *global*. Il permet de préciser qu'une variable appartient à l'espace mémoire général. Il doit être utilisé au début d'une fonction et permet de préciser quelles variables sont globales et surtout lesquelles sont accessibles par la fonction.

Code : Python

```
def fonction1() :
    """Fonction de test 1"""
    global variable
    variable = 10
    variable **= 2
    print("Variable = %f" % variable)

def fonction2() :
    """Fonction qui devrait maintenant fonctionner."""
    global variable
    print("Variable = %f" % variable)

fonction1()
fonction2()
```

Ce code devrait maintenant fonctionner. En effet, le mot clé *global* permet aux fonctions de reconnaître *variable*. Cependant, ce genre de pratique n'est pas très recommandée ! En effet, le fait que chaque fonction possède son espace mémoire évite d'affecter des variables sans le vouloir. Il vaut donc mieux créer des fonctions qui retournent un résultat plutôt que des procédures qui modifient directement la valeur de nos variables.

Paramètres des fonctions

Nous avons vu comment définir une fonction et en définir les arguments assez rapidement. Mais la déclaration de fonctions offrent quelques spécificités intéressantes. C'est ce que nous allons voir tout de suite !

Valeurs par défaut

En Python, il est possible de définir des valeurs par défaut aux arguments des fonctions. Et c'est d'ailleurs assez simple à faire, il suffit d'affecter une valeur aux arguments directement dans la déclaration de la fonction. Faisons un essai :

Code : Python

```
def puissance(nombre, puissance = 2) :
    """
Fonction permettant de calculer la puissance d'un nombre.
nombre : Le nombre dont on veut la puissance.
puissance = 2 : La puissance à laquelle on souhaite passer nombre
return : Retourne nombre ** puissance
"""
    return nombre ** puissance

print(puissance(3,3))
print(puissance(3))
```

Dans le premier cas, nous appelons la fonction puissance en précisant les deux arguments, dans le second cas, la puissance n'est pas précisée et est automatiquement définie à la valeur 2 donnée dans la définition de la fonction. Le fait de définir des valeurs par défaut permet alors de ne pas donner tous les arguments possibles d'une fonction ! Bien entendu, tous les types de données peuvent être utilisés comme valeur **par défaut et aucune restriction n'est faite ici. La seule contrainte réside dans le fait que les arguments possédant une valeur par défaut doivent nécessairement être déclarés après ceux qui n'en possèdent pas** (arguments obligatoires) dans la déclaration de la fonction. Ainsi, vous avez une déclaration de fonction nécessairement de cette forme :

Code : Python

```
def fonction(arg1, arg2, arg3 = valeur, arg4 = valeur2, ...) :
    """Documentation"""
    instructions...
```

Appel désordonné

Il peut arriver lorsque l'on programme de devenir un peu dérangé.  Heureusement, Python vous propose pour ça une fonctionnalité intéressante : l'appel aux fonctions avec des arguments désordonnés. Cela vous permet de donner les arguments d'une fonction dans n'importe quel ordre. Pour faire cela, il suffit de préciser le nom des arguments que l'on donne aux fonctions :

Code : Python

```
def puissance(nombre, puissance = 2) :
    """
Fonction permettant de calculer la puissance d'un nombre.
nombre : Le nombre dont on veut la puissance.
puissance = 2 : La puissance à laquelle on souhaite passer nombre
return : Retourne nombre ** puissance
"""
    return nombre ** puissance

print(puissance(puissance = 4, nombre = 3))
```

Vous voyez ici que j'inverse l'ordre des arguments de la déclaration de ma fonction lorsque je l'appelle. Pour cela, je précise à quel argument j'affecte quelle valeur. Ce genre d'appel est possible avec un nombre indéterminé d'argument et permet quand même de ne pas préciser de valeur pour les paramètres facultatifs.

Il est aussi possible de mélanger les deux types d'appels de fonction en donnant les premiers arguments de façon ordonnée puis de mélanger pour la suite mais les arguments ordonnés doivent impérativement être donnés en premier !

Code : Python

```
def operations(nombre, plus = 2, moins = 2, divise = 3, multiplie =
5) :
    """ Fonction stupide qui ne présente aucun intérêt.
nombre : Un nombre sur lequel on veut effectuer des opérations.
plus : Un nombre à additionner à nombre.
divise : Un nombre pour diviser nombre.
multiplie : Un nombre pour multiplier nombre.
return : Un tuple (nombre+plus, nombre-moins, nombre/divise,
nombre*multiplie)"""
    return nombre+plus, nombre-moins, nombre/divise,
nombre*multiplie

print(operations(4,4, divise = 6, multiplie = 8))
```

Nombre indéfini d'arguments

Il existe des fonctions qui acceptent un nombre indéterminé d'arguments, c'est le cas de la fonction *print* par exemple. Comment cela fonctionne-t-il ? Et bien, c'est assez simple. Pour cela, il est possible de le faire de deux manières : avec des arguments anonymes ou avec arguments associés à des clés. Nous allons donc voir les deux manières de créer des fonctions avec un nombre indéterminé de paramètres.

Arguments anonymes

La syntaxe est particulièrement simple. En fait, vous allez préciser une séquence (liste, tuple ou chaîne de caractères) comme argument. Le fait que cet argument doive être une séquence se précise par l'utilisation de l'opérateur '*' juste avant le nom de l'argument :

Code : Python

```
def fonction(*arguments) :
    """Test de fonction avec un nombre indéfini d'arguments.
arguments : Une séquence à écrire en console."""
    for element in arguments :
        print(element)

fonction(43, 38, "Peuh !", True)
```

De cette façon, il est donc possible d'appeler ma fonction avec un nombre indéfini d'arguments mais attention, cela est possible à une seule condition : la séquence d'arguments doit impérativement se situer en dernier argument de la fonction ! Les arguments s'écrivent donc dans l'ordre suivant dans la déclaration d'une fonction : **les arguments obligatoires, les arguments avec valeur par défaut (facultatifs) et enfin des arguments supplémentaires si nécessaires.**

Il est tout à fait possible de passer pour ce type d'argument une séquence, pour cela, il suffit de la faire précéder d'une étoile pour la convertir.

Code : Python

```
def fonction(*arguments) :
    """Test de fonction avec un nombre indéfini d'arguments.
arguments : Une séquence à écrire en console."""
    for element in arguments :
        print(element)

fonction(*'abcdef')
```

Si vous n'utilisez pas l'opérateur '*', votre séquence sera alors considérée comme n'étant qu'un seul argument ! Cela est dû au typage dynamique. En effet, n'importe quel argument de votre fonction peut prendre n'importe quel type, il faut donc préciser ici que chaque élément de la séquence est un argument à part entière.

Arguments avec clé

Si vous voulez associer une clé à chacun des arguments que vous donnez en supplément à une fonction, il va falloir utiliser un dictionnaire. Et ce coup-ci, c'est l'opérateur '**' qui sera utilisé.

Code : Python

```
def fonction(**arguments) :
    """Test de fonction avec un nombre indéfini d'arguments.
arguments : Un dictionnaire à écrire en console."""
    for cle in arguments :
        print(cle,arguments[cle])

fonction(arg1 = 'Peuh !', arg2 = 38)
```

Vous verrez en exécutant ce code que l'on peut donc facilement créer des couples (clé,valeur) à passer en paramètre. La contrainte est la même que précédemment et le dictionnaire d'arguments doit être impérativement placé en dernier dans la liste d'arguments lors de la déclaration de la fonction. Encore une fois, pour convertir un dictionnaire non pas un seul argument mais en une suite d'arguments, il faut utiliser l'opérateur '**' devant le dictionnaire passé en argument.

Code : Python

```
def fonction(**arguments) :
    """Test de fonction avec un nombre indéfini d'arguments.
arguments : Un dictionnaire à écrire en console."""
    for cle in arguments :
        print(cle,arguments[cle])

fonction(**dict(arg1 = 'Peuh !', arg2 = 38))
```

Vous savez créer des fonctions et y faire appel. Dans la prochaine partie, nous verrons comment créer ses propres modules pour organiser son code mais aussi un type de fonction très particulière en Python : les fonctions lambda. Enfin, vous pourrez voir une application assez pratique des fonctions avec la récursivité !

Les fonctions (2/2)

Maintenant que vous avez vu comment définir et utiliser des fonctions, nous allons voir comment aller un peu plus loin dans les fonctions ! La programmation modulaire vous sera sûrement utile, il faut donc que vous appreniez à créer des modules que vous et vos amis pourrez importer dans tous vos programmes.

Ensuite, nous verrons quelques pratiques spécifiques aux fonctions avec les fonctions lambda définies à la volée et les fonctions récursives qui vous permettront de jolis effets !

Si vous n'êtes pas très à l'aise avec ce que nous avons vu jusqu'à maintenant, n'hésitez surtout pas à vous entraîner car ce chapitre ne sera pas parmi les plus simples. Au boulot !

Créer des modules

Rappel

Nous avons vu dans le prélude sur les fonctions qu'il en existe des milliers déjà implémentées dans Python que vous pouvez importer dans vos programme. Pour cela vous pouvez utiliser deux types de phrases :

Code : Python

```
from nomDuModule import nomDeLaFonction
```

OU

Code : Python

```
import nomDuModule
```

De cette manière vous pouvez accéder à de nombreuses fonctionnalités. Mais vous vous demandez certainement s'il est possible d'importer des modules que l'on aurait créés soi-même. Et bien oui ! L'importation se fait en fait exactement de la même manière, il suffit simplement de préciser pour nom de module le nom de votre fichier (sans l'extension .py) contenant les fonctions que vous voulez.

N'oubliez pas le caractère '*' qui veut dire 'tout' si vous voulez importer toutes les fonctions d'un module.

Créer un module

Faisons un petit test. Nous allons créer un fichier monModule.py avec quelques fonctions :

Code : Python - monModule.py

```
def foo() :
    print("foo")

def bar() :
    print("bar")

def foobar() :
    print("foobar")

def barfoo() :
    print("barfoo")
```

Je crée donc quatre fonctions totalement inutiles mais nous allons essayer de les importer.

Je vous avais épargné jusqu'à maintenant les exemples à coup de foo et de bar... Sachez qu'il est de tradition dans l'informatique d'utiliser ces deux noms ou une combinaison de ces noms. Ce sont des noms génériques ou variables

métasyntaxiques.

Si nous voulons importer notre module, dans un autre fichier .py, essayez d'utiliser la même syntaxe que pour les modules intégrés dans Python.

Code : Python

```
from monModule import *

foo()
bar()
foobar()
```

OU

Code : Python

```
import monModule

monModule.foo()
monModule.bar()
monModule.foobar()
```

Vous devriez voir apparaître la même chose dans les deux cas. En effet, ces deux écritures sont équivalentes. La deuxième, plus lourde, évite les ambiguïtés notamment si plusieurs modules possèdent des fonctions avec le même nom. Cela peut arriver plus souvent que vous ne l'imaginez.

La programmation modulaire vous permettra de créer des modules cohérents contenant des fonctions qui s'appliquent toutes dans un contexte particulier. Par exemple, si vous voulez créer un ensemble de fonctions qui effectuent des opérations mathématiques, il serait bon de toutes les regrouper dans un module spécifique. C'est ce que fait le module *math* intégré à Python par exemple. En consultant un peu la documentation, vous remarquerez très vite que chaque module correspond à un thème ou à un type d'application spécifique. Essayez donc de structurer vos programmes de la même façon. De plus, les modules que vous créerez ainsi pourront être réutilisés dans tous vos programmes si besoin est !

Différencier le programme

Maintenant que vous savez que vous pouvez utiliser plusieurs fichiers pour écrire vos programmes en utilisant ce que l'on appelle la programmation modulaire, il faut que vous sachiez comment l'interpréteur différencie ces programmes. En effet, nous allons faire un petit test qui va vous paraître bête. Dans votre module, rajoutez les lignes :

Code : Python

```
foo()
bar()
foobar()
```

En console, vous devriez voir apparaître :

Code : Console

```
>>>
foo
bar
foobar
foo
bar
foobar
```

Mais que se passe-t-il ?! Le code présent dans votre module s'exécute au moment de l'importation ! Mais il est extrêmement rare que je veuille faire une telle chose ! En fait, lorsque vous importez un module dans votre code, l'interpréteur ne se crée qu'un seul fichier avec l'ensemble du code de vos différents modules dans un seul fichier. C'est ce fichier qu'il va interpréter... Seulement, vous pouvez développer des modules avec de nombreuses fonctions et du code de test de ces fonctions en fin de fichier par exemple. Et ce code n'a que pour seul intérêt de tester le bon fonctionnement de vos fonctions. Pour résoudre ce problème, il existe des variables qui existent toujours en Python. Celle qui va nous intéresser ici s'appelle *__name__*.

En Python, toutes les variables internes au langage lui-même et qui seront toujours présentes commencent et finissent toutes par un double 'underscore' (tiret bas). Il en existe des dizaines qui vous permettent d'accéder à certains informations, vous trouverez plus d'informations à propos de ces variables dans la documentation.

Dans notre cas, nous allons vérifier que le module que l'on veut importer est le programme principal ou non. Pour cela, il faut faire un test sur la variable *__name__*.

Code : Python - monModule.py

```
def foo() :
    print("foo")

def bar() :
    print("bar")

def foobar() :
    print("foobar")

def barfoo() :
    print("barfoo")

if __name__ == "__main__" :
    foo()
    bar()
    foobar()
```

Avec ce test, le code qui suit ne sera considéré par l'interpréteur que si le fichier monModule.py est le programme principal exécuté (*main*). Sinon, je ne fais rien. Essayez de lancer votre fichier monModule.py puis votre autre programme important monModule.py. Vous verrez alors que l'appel aux fonctions effectué dans monModule.py ne se fera que si c'est ce fichier qui est directement exécuté.

Comme j'en parlais tout à l'heure, avec ce test vous pouvez alors créer des modules exécutables pour tester l'ensemble de leurs fonctions mais qui ne font rien lorsqu'on les importe !

Fonction lambda

Une des particularités de Python est les fonctions déclarées à la volée aussi appelées fonctions lambda. Ces fonctions seront en fait déclarées en plein milieu de votre code et seront anonymes. Pour les déclarer, le mot clé *def* ne sera pas nécessaire, vous devrez utiliser le mot clé *lambda*.

Ces fonctions portent le nom de lambda car c'est le nom générique pour désigner un anonyme. Vous avez certainement déjà entendu parler de l'individu lambda. Il ne représente personne en particulier !

Nous allons voir tout de suite la syntaxe générale d'une fonction lambda :

Code : Python

```
lambda arg1, arg2 ... : instructions
```

Pour déclarer une fonction lambda, il faut donc utiliser le mot clé lambda, le faire suivre d'arguments ou pas que la fonction prendra en compte et enfin des instructions à exécuter. Si vous voulez pouvoir appeler votre fonction lambda, il faut la stocker dans une variable qui pourra donc être appelée comme n'importe quelle autre fonction :

Code : Python

```
aLArrache = lambda nombre, puissance : nombre**puissance
```

```
print(aLArrache(2,4))
```

Avec ce petit exemple, vous voyez qu'il est possible de déclarer une fonction à la volée et de l'appeler ensuite.

Remarquez ici que le mot clé *return* n'est pas nécessaire dans les instructions d'une fonction lambda aussi simple. Les fonctions lambda seront d'ailleurs souvent extrêmement simple et ne serviront qu'à faire quelques opérations de bases.

Bien sûr, vous devez vous demander à quoi ces fonctions peuvent bien servir... Et bien, vous pouvez faire des fonctions qui vous retourne des fonctions lambda générées en fonction de ce que vous passez en argument par exemple ! Mais nous allons voir qu'il existe des cas où les fonctions lambda sont très utiles !

Utilité

En Python, il existe plusieurs fonctions intégrées qui s'appliquent aux fonctions. Et oui, il est possible d'appliquer des fonctions à d'autres fonctions comme si c'était de simples variables ! Les trois qui vont nous intéresser sont les suivantes :

- **filter** (filtrer): s'applique aux fonctions booléennes (qui retournent *True* ou *False*). Applique une séquence d'arguments à la fonction ainsi testée et retourne la séquence d'arguments pour lesquels elle a retournée *True*.
- **map** (carte) : applique la fonction passée en argument à une ou plusieurs séquences passées en argument. Retourne un ensemble de tuples correspondant aux résultats des appels de la fonction.
- **reduce** (réduire) (cette fonction a été retirée en Python v3) : applique une fonction binaire (c'est-à-dire qui accepte deux arguments) aux éléments d'une séquence les uns à la suite des autres pour réduire le résultat à une seule valeur de retour. Une valeur initiale peut être spécifiée.

Expliqué comme ça, ça ne vous parle certainement pas beaucoup. Nous allons donc voir quelques petits exemples avec des fonctions lambda.

filter() :

Cette fonction a pour but de séparer les éléments d'une séquence qui retourneront *True* pour ne vous retourner que ces derniers. Vous avez par exemple une séquence de données sur lesquelles vous allez effectuer des tests pour savoir si elles sont utilisables ou non. Vous pouvez le faire en une seule ligne à l'aide d'une fonction lambda et de la fonction *filter*.

Code : Python

```
>>> from random import randint
>>> aleatoires = []
>>> for i in range(100) :
 aleatoires.append(randint(0,100))

>>> filter(lambda nombre : nombre%3, aleatoires)
<filter object at 0x01F87490>
>>> list(filter(lambda nombre : nombre%3, aleatoires))
[52, 97, 32, 86, 14, 98, 8, 65, 86, 11, 11, 38, 74, 88, 46, 19, 7,
95, 19, 22, 26, 68,
5, 32, 97, 7, 47, 64, 65, 85, 47, 38, 91, 5, 20, 88, 35, 61, 83, 10,
49, 10, 88, 83, 46,
89, 55, 92, 40, 49, 44, 40, 77, 61, 53, 52, 59, 43, 98, 11, 86, 8,
58, 13, 61, 86]

>>> list(filter(lambda nombre : not nombre%3, aleatoires))
[45, 15, 39, 18, 9, 9, 12, 45, 93, 96, 87, 75, 87, 99, 96, 69, 93,
12, 51, 30, 18, 39,
57, 36, 87, 78, 87, 84, 36, 51, 72, 24, 90, 96]
```

Dans cet exemple, j'importe la fonction *randint* qui permet de générer des nombres entiers pseudo-aléatoires. Je crée ensuite une liste de 100 nombres aléatoires compris entre 0 et 100. Je souhaite ensuite appliquer un filtre sur ces nombres, c'est pourquoi j'utilise la fonction *filter*.

J'applique un filtre qui doit me donner tous les nombres qui ne sont pas divisibles par 3. Souvenez vous que les booléens peuvent être remplacés par des nombres où 0 vaut *False* et tous les autres valent *True*. Remarquez que cette fonction renvoie un objet de type *filter* que je passe en argument de la fonction *list* uniquement pour l'afficher correctement (l'emplacement en

mémoire indiqué en hexadécimal ne m'est pas d'une grand utilité). Dans le deuxième cas, je cherche tous les nombres de la liste qui sont des multiples de 3. De façon générale, la fonction *filter* s'écrit :

Code : Python

```
filter(fonction, sequence)
```

Je lui ai donc passé ici une fonction lambda en argument et ma séquence de nombre aléatoires. De cette façon, je génère donc rapidement des filtres sans avoir à déclarer de fonction qui ne sera certainement pas utile ailleurs que dans ce cas précis.

map() :

La fonction *map* permet de créer littéralement la carte des résultats de la fonction passée en argument. Si vous lui passez plusieurs séquences en paramètres, cela vous retournera le résultat pour l'ensemble des couples ou n-uplés d'arguments, la correspondance se faisant à indices égaux. En clair, vous devez utiliser un nombre de séquences égal au nombre d'arguments que prend votre fonction passée en argument et cela vous retournera la liste des résultats de la taille de la plus petite séquence d'arguments donnée. Voyons un peu plus concrètement ce que cela donne :

Code : Python

```
>>> liste = list(range(10))
>>> resultats = map(lambda x, y : x + y, liste, liste)
>>> for element in resultats :
 print(element)

0
2
4
6
8
10
12
14
16
18
```

Ici, je crée d'abord une liste des nombres de 0 à 9. Je cherche ensuite à effectuer la carte des résultats et je l'imprime. Vous voyez ici que les couples d'arguments formés sont (0,0), (1,1), ... , (9,9). Ceci explique ce que j'affiche. En utilisant les fonctions lambda pour définir une fonction directement dans *map*, je m'évite encore une fois une déclaration futile de fonction.

reduce() (non valide en Python v3) :

La fonction reduce s'applique uniquement aux fonctions binaires, c'est-à-dire qui accepte deux arguments. Le principe est simple, vous appliquez la fonction passée en argument au résultat de celle-ci avec l'élément suivant dans la séquence passé en paramètre. Si ce n'est pas clair, voyons ça avec un exemple :

Code : Python

```
>>> liste = list(range(1,100))
>>> reduce(lambda x, y : x*y, liste)
933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000
```

Ici, ce que j'ai fait permet de calculer le factoriel d'un nombre.

Pour ceux d'entre vous qui ne connaissent pas l'opération factorielle, c'est une opération mathématique noté 'unNombre!' et qui consiste à calculer le produit de unNombre * (unNombre - 1) * (unNombre - 2) * ... * 3 * 2 * 1. C'est l'opération qui croit le plus vite, et comme vous pouvez le constater, 99! est particulièrement énorme !

Dans cet exemple, j'applique donc la fonction multiplication aux deux arguments de ma fonction lambda et *reduce* permet de l'appliquer successivement à 1 * 2 puis au résultat de cette opération fois 3 et ainsi de suite. Malheureusement, cette manière de faire n'étant que partiellement lisible, la fonction *reduce* a été retirée du langage Python en version 3. En effet, il est possible d'effectuer la même chose avec une simple boucle *for* beaucoup plus facile à comprendre...

Code : Python

```
>>> factoriel = 1
>>> for nombre in range(1,100) :
 factoriel *= nombre

>>> factoriel
933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000
```

Ceci est bien entendu beaucoup plus lisible et plus simple à comprendre ! Vous comprendrez donc pourquoi cette fonction a été supprimée.

Récursivité

Une application très intéressante des fonctions est ce que l'on appelle la récursivité. Ce mot qui fait un peu peur exprime un concept en réalité assez simple : une fonction est dite récursive lorsqu'elle fait appel à elle-même.

Quoi ?! Une fonction qui s'appelle elle-même ?! Mais je vais faire une boucle infinie avec ça ! 

Oui ! Si vous n'introduisez pas de condition d'arrêt ! Et c'est là toute la difficulté des fonctions récursives. Il faut réussir à exprimer le cas dit trivial, évident, celui qui ne nécessite pas un nouvel appel de la fonction. Mais si je ne vous donne pas un exemple, je suis certain que vous ne verrez pas du tout l'utilité d'une fonction récursive...

Code : Python

```
def factoriel(nombre) :
    """
Cette fonction retourne le factoriel d'un nombre.
nombre : Le nombre dont on veut le factoriel.
return : nombre!
"""
    if nombre == 1 or nombre == 0 :
        return 1
    else :
        return nombre * factoriel(nombre - 1)

print("2! = %i" % factoriel(2))
print("3! = %i" % factoriel(3))
print("4! = %i" % factoriel(4))
print("5! = %i" % factoriel(5))
print("6! = %i" % factoriel(6))
print("7! = %i" % factoriel(7))
```

La fonction factorielle est en effet par définition une fonction récursive. Comme nous l'avons vu plus haut, il est tout à fait possible de le calculer avec une simple boucle *for* mais le fait que $N! = N * (N-1)!$ nous donne ici une excellente occasion d'introduire les fonctions récursives. De plus, la condition d'arrêt, le cas trivial est donné par le fait que 1! = 0! = 1, cela fait partie de la définition mathématique de l'opération factorielle. Le cas trivial est donc bien défini pour arrêter l'appel récursif.

Vous êtes limité dans le nombre de fonctions que vous pouvez appeler ! Si trop de fonctions sont appelées sans avoir été résolues (terminées), l'interpréteur vous fera une jolie erreur. L'interpréteur Python est même capable de vous signaler que vous demandez une récursion trop importante. Essayez donc de calculer le factoriel de 5000 avec la fonction précédente. Vous devriez obtenir une erreur de ce style :

Code : Console

```
  File "G:\Travail\Site du zÃ©ro\Langage Python\Exemples\factoriel.py", line 7,
```

```
RuntimeError: maximum recursion depth exceeded in cmp
```

Même si l'utilité de telles fonctions ne vous semble pas évident, vous serez certainement confronté à un problème y faisant appel par nature un jour ou l'autre. Souvenez vous simplement qu'il faut bien définir une condition d'arrêt à votre fonction.
Vous êtes maintenant des pros des fonctions ! Vous savez structurer votre code, l'organiser et le rendre de plus en plus lisible. Il ne reste donc qu'un dernier point à voir avant de nous attaquer à de vrais projets concrets qui vous donneront du fil à retordre !

Gérer les exceptions

Des fois, on voudrait pouvoir essayer de faire quelque chose dans un programme mais ça ne fonctionne pas. Et malheureusement, notre programme est interrompu et ce n'est pas du tout ce que l'on voulait ! Mais alors, est-il possible de gérer ce genre de problèmes ? Oui ! Ca s'appelle la gestion des exceptions !

Ici, vous verrez ce qu'est une exception, comment les gérer et à quoi cela peut servir de les gérer !

Les exceptions

Qu'est-ce qu'une exception ? Une exception est ce qui se produit dans votre programme lorsque celui-ci sort du flux d'instructions habituel. Lorsque votre programme rencontre une erreur, il commence par planter, c'est-à-dire qu'il sort du flux d'instructions et ensuite il fait une exception, il va donc chercher d'où vient l'erreur.

Les exceptions sont ce que vous affiche l'interpréteur lorsque vos programmes ne fonctionnent pas correctement. Il vous indique où se trouve l'erreur et pourquoi elle a lieu. En Python, il existe un moyen de gérer les exceptions pour que vos programmes ne plantent plus à chaque fois. Pour cela, il suffira de lui proposer une suite d'instructions à exécuter en cas d'erreur.

Les exceptions les plus fréquentes que vous rencontrerez sont les suivantes :

- **NameError** : cela apparaît lorsque vous essayez de manipuler une variable ou une fonction non déclarée.
- **ZeroDivisionError** : cela apparaît lorsque vous essayez de diviser par zéro (et oui, c'est impossible !).
- **SyntaxError** : cela apparaît lorsque vous avez mal écrit une structure ou que vos parenthèses ne sont pas correctes.
- **IndexError** : cela apparaît lorsque vous essayez d'accéder à un élément à un indice qui n'existe pas dans une séquence.
- **KeyError** : cela apparaît lorsque vous essayez d'accéder à un élément d'un dictionnaire avec une clé qui n'existe pas.
- **IOError** : cela apparaît lorsque vous manipulez mal un fichier comme essayer de lire un fichier qui n'existe pas ou d'écrire un fichier ouvert en lecture uniquement.
- **TypeError** : cela apparaît lorsque vous essayez d'effectuer des opérations entre des types incompatibles (ex : 12 + 'peuh !').

Bien, assez parlé, passons à la suite voyons comment il est possible de gérer les exceptions.

Lever une exception

Il existe une structure spécifique en Python qui permet de lever des exceptions. Elle se construit à l'aide des mots clés *try* et *except*. Il existe d'autres manières de faire que nous allons aussi voir.

Structure minimale

La structure minimale pour gérer les exceptions s'écrit de la façon suivante :

Code : Python

```
try :
    instructions à risque...
except :
    à faire en cas d'erreur...
```

J'avais déjà utilisé cette structure dans le corrigé du TP sur le Juste Prix. Elle me permettait dans cette exemple de gérer une mauvaise entrée de l'utilisateur lorsque je lui demande d'écrire un nombre. Par exemple :

Code : Python

```
try :
    nombre = input("Entrez un nombre : ")
    nombre = int(nombre)
    print("Vous avez bien tapé un nombre !")
except :
    print("Vous auriez dû taper un nombre...")
```

Si vous essayez ce code, vous pouvez vous apercevoir que la suite d'instructions dans la structure *try* est immédiatement interrompue lorsqu'une erreur se produit. Ceci entraîne que je n'écris pas "Vous avez bien tapé un nombre !" lorsque ce n'est pas le cas ! Vous avez donc ici la structure minimale de la gestion des exceptions.

Des types d'erreurs

Il est tout à fait possible de provoquer un comportement différent en fonction de l'exception qui est levée. Pour cela, il suffit de le préciser après *except*. Et il est même possible de proposer plusieurs alternatives en fonction de l'erreur provoquée. Prenons un petit exemple simple :

Code : Python

```
nombre = input("Entrez un nombre : ")
try :
    nombre = int(nombre)
    print("Vous avez bien tapé un nombre !")
    print("Essayons de diviser par ce nombre...")
    exemple = 10/nombre
    print("Nous obtenons : 10/%i = %f" % (nombre, 10/nombre))
except ValueError :
    print("Vous auriez dû taper un nombre...")
except ZeroDivisionError :
    print("Vous avez entré le nombre 0, il n'est pas possible de
diviser par 0...")
```

Dans cet exemple, deux types d'exceptions peuvent être levées et ne donneront pas le même résultat. J'ai décidé d'écrire dans la console un texte correspondant au type d'erreur !

Il est aussi possible d'effectuer la même suite d'instructions pour différentes erreurs, pour cela, il vous suffit de les inclure dans un tuple à la suite d'*except* :

Code : Python

```
nombre = input("Entrez un nombre : ")
try :
    nombre = int(nombre)
    print("Vous avez bien tapé un nombre !")
    print("Essayons de diviser par ce nombre...")
    exemple = 10/nombre
    print("Nous obtenons : 10/%i = %f" % (nombre, 10/nombre))
except (ValueError, ZeroDivisionError, TypeError) :
    print("Une erreur est survenue...")
```

Si vous n'avez aucune idée de toutes les exceptions qui peuvent intervenir dans votre portion de code ou si tout simplement vous ne souhaitez pas effectuer des instructions différentes pour toute une catégorie d'exceptions, vous pouvez vous contenter de ne préciser aucune exception en particulier à *except*.

Code : Python

```
nombre = input("Entrez un nombre : ")
try :
    nombre = int(nombre)
    print("Vous avez bien tapé un nombre !")
    print("Essayons de diviser par ce nombre...")
    exemple = 10/nombre
    print("Nous obtenons : 10/%i = %f" % (nombre, 10/nombre))
except ValueError :
    print("Vous n'avez pas entré un nombre...")
except :
    print("Une erreur est survenue...")
```

Exécuter selon le cas

Lorsque l'on gère des exceptions, il arrive que l'on veuille exécuter des instructions selon certaines conditions de réussite ou non

des instructions précédentes. Nous venons de voir que si une erreur se produit, alors les instructions dans la structure *try* sont interrompues et le flux d'instructions se dirige alors vers un bloc *except*. Seulement, il se peut qu'une fois nos actions dangereuses effectuées, plus rien qui puisse provoquer une erreur n'ait à être exécuté. Pour cela, il existe deux mots clés : *else* qui vous permettra ici d'exécuter des instructions si tout s'est bien déroulé et *finally* qui s'exécutera quoi qu'il arrive.

Code : Python

```
nombre = input("Entrez un nombre : ")
try :
    nombre = int(nombre)
    print("Vous avez bien tapé un nombre !")
    print("Essayons de diviser par ce nombre...")
    exemple = 10/nombre
    print("Nous obtenons : 10/%i = %f" % (nombre, 10/nombre))
except ValueError :
    print("Vous n'avez pas entré un nombre...")
except :
    print("Une erreur est survenue...")
else :
    print("Tout s'est bien déroulé !")
finally :
    print("Je suis le bloc de fin !")
```

Dans cet exemple, la gestion des exceptions est complète. Ainsi, il est possible de gérer certains types d'exceptions. Vous pouvez vérifier que le bloc *else* ne s'exécute qu'en cas de bon déroulement du programme et le bloc *finally*, lui, s'exécute dans tous les cas, qu'il y ait une erreur ou non.

Interrompre le programme

Vous savez gérer vos exceptions. Seulement, lorsque vous les gérez, il se peut que vous oubliez quelques cas et peut-être aimeriez-vous être au courant que votre programme n'a effectivement pas fonctionné correctement et pourquoi. Et bien, même en utilisant une structure *try-except*, il est tout à fait possible de lever une exception et d'interrompre le programme. Un mot clé vous permet d'effectuer cette action : *raise* (lever).

Code : Python

```
nombre = input("Entrez un nombre : ")
try :
    nombre = int(nombre)
    print("Vous avez bien tapé un nombre !")
    print("Essayons de diviser par ce nombre...")
    exemple = 10/nombre
    print("Nous obtenons : 10/%i = %f" % (nombre, 10/nombre))
except ValueError :
    print("Vous n'avez pas entré un nombre...")
except :
    print("Une erreur est survenue...")
    raise
else :
    print("Tout s'est bien déroulé !")
finally :
    print("Je suis le bloc de fin !")
```

Par exemple, ici, je construis la structure complète mais ne gère réellement que les erreurs de type *ValueError*. Pour les autres, je me contentais d'écrire qu'une erreur était survenue. Mais si je veux en connaître un peu plus sur l'erreur, je dois lever une exception, et c'est ce que fait le mot clé *raise*. Ce mot clé interrompt le programme et écrit dans la console l'erreur qui est apparue.

Les assertions

Il existe une autre manière de lever des exceptions en Python à l'aide du mot clé *assert*. Cette structure se compose de ce mot clé suivi d'une expression qui renvoie un booléen. Si celui-ci est vrai, alors rien ne se passe, si celui-ci est faux, alors une exception du type *AssertionError* est levée.

Cette structure se construit donc de cette manière :

Code : Python

```
assert expressionLogique
```

Par exemple, vous pouvez avoir quelque chose de la forme :

Code : Python

```
assert 2 + 2 == 2 * 2
assert len(list(range(10)) <= 10
```

Dans ces deux cas, aucune exception ne sera levée. Cette structure vous permet d'assurer que certaines conditions sont bien vérifiées et ont tout intérêt à être placées au sein de structures du type *try-except*.

Code : Python

```
nombre = input("Entrez un nombre : ")
try :
    nombre = int(nombre)
    print("Vous avez bien tapé un nombre !")
    print("Essayons de diviser par ce nombre...")
    assert nombre != 0
    exemple = 10/nombre
    print("Nous obtenons : 10/%i = %f" % (nombre, 10/nombre))
except ValueError :
    print("Vous n'avez pas entré un nombre...")
except AssertionError :
    print("Il est impossible de diviser par zéro...")
except :
    print("Une erreur est survenue...")
    raise
```

Je vérifie ici, avant d'essayer de faire un calcul dangereux, si le nombre entré par l'utilisateur n'est pas égale à 0. S'il vaut 0, alors une exception est levée. De cette manière, je gère de ma propre manière une erreur qui serait intervenue et l'empêche d'apparaître ! Les erreurs doivent le plus souvent possible être gérées pour ne laisser aucun plantage face à l'utilisateur finale et permettre de diagnostiquer au mieux ce qui s'est mal passé. Vous devrez apprendre à faire avec pour propager correctement les erreurs d'une fonction à l'autre par exemple et à faire en sorte que votre programme soit capable de déterminer la provenance exacte de l'erreur. Bien entendu, la documentation des erreurs possiblement levées par vos différentes fonctions sera une première voie vers une bonne compréhension par les autres de ce que fait votre code !

Maintenant que vous avez acquis les concepts de base de la programmation en Python et même de la programmation en général, il est temps de passer à la vitesse supérieure avec la programmation orientée objet. Vous verrez que c'est un concept qui n'est pas toujours évident à comprendre mais qui a prouvé son efficacité !

Mais avant, je vous propose de vous exercer un peu avec de petits projets qui pourront vous prendre plusieurs heures chacun à développer !

TP 2 : Jeu de la Vie

Vous avez maintenant de bonnes bases théoriques sur la programmation fonctionnelles mais maintenant, vous allez devoir vous exercer ! Pour cela, je vous propose un petit TP qui vous prendra certainement quelques heures si vous le faites entièrement de vous-même.

Ici, je vais essayer de faire un corrigé progressif qui vous permette de vous débloquer lorsque vous n'y arrivez pas ! Le but est donc que vous soyez laisser seul face à votre éditeur de texte dans un premier temps. Ensuite, vous pourrez utiliser le corrigé comme un guide lorsque vous être bloqué.

Je sens que vous êtes impatients de connaître le sujet du TP alors ne perdons pas de temps !

Présentation du sujet

Le jeu de la vie, contrairement à son nom, n'est pas du tout un jeu ! C'est un programme de simulation de vie primitive qui se propage selon des règles précises. En fait, la simulation se déroule sur une grille. Chaque case peut être dans un état mort ou vivant. La simulation se fait de façon itérative, c'est-à-dire que l'on va effectuer des modifications de la grille avec des pas discrets (correspondant à un temps fixe arbitraire). Dans les règles de bases, d'un pas sur l'autre, chacune des cases (ou cellules) se transforment comme ceci :

- Si une cellule possède exactement trois voisines vivantes, elle est vivante à l'itération suivante.
- Si une cellule possède exactement deux voisines vivantes, elle ne change pas d'état à l'itération suivante.
- Dans les autres cas, elle est morte à l'itération suivante.

Le but de ce petit automate est de réussir à trouver des configurations persistantes qui vont créer un motif répété. L'animation ci-dessus en est un excellent exemple avec comme un tir de canon de petits avions... Vous pourrez trouver des informations à ce sujet dans l'article wikipédia.

Quelques idées

Le projet en lui-même n'est vraiment pas compliqué ! Cependant, si l'on veut s'amuser un peu, il est possible de proposer pas mal d'options différentes :

- Différentes tailles de grille
- Différentes configuration initiale (forme d'animaux, etc...)
- Chercher automatiquement des formes stables ou périodiques
- Mesure de durée de vie d'un système
- Statistiques sur des configurations
- Proposer d'autres règles

Ce sujet qui paraît simple au départ offre de nombreuses possibilités et suggère un travail d'étude intéressant une fois le projet implémenté. Le moteur de la simulation est simple mais les options que vous proposerez représenteront donc tout l'intérêt de votre programme.

Pour ce qui est du résultat, je vous donne quelques petites "captures d'écran" de ce que j'obtiens.

Code : Console - Menu principal

```
.
      JJJJ  EEEEEE  UU      UU       DDDDD    EEEEEE
      JJJJ  EEEEEE  UU      UU       DDDDDD   EEEEEE
       JJ   EE      UU      UU       DD   DD  EE
       JJ   EE      UU      UU       DD   DD  EE
```

```
      JJ   EEEE    UU     UU      DD   DD  EEEE
  JJJJ JJ   EEEE    UU     UU      DD   DD  EEEE
  JJJJ JJ   EE      UU     UU      DD   DD  EE
   JJ  JJ   EE      UU     UU      DD   DD  EE
   JJJJJJ   EEEEEE  UUUUUUUUU      DDDDDD   EEEEEE
    JJJJ    EEEEEE   UUUUUUU       DDDDD    EEEEEE


 LL       AAAAAA       VV         VV  IIIIII  EEEEEE
 LL      AAAAAAAA      VV         VV  IIIIII  EEEEEE
 LL      AA    AA      VV         VV    II    EE
 LL      AA    AA      VV         VV    II    EE
 LL      AA    AA      VV         VV    II    EEEE
 LL      AAAAAAAA       VV       VV     II    EEEE
 LL      AAAAAAAA        VV     VV      II    EE
 LL      AA    AA         VV   VV       II    EE
 LLLLLL  AA    AA          VV VV      IIIIII  EEEEEE
 LLLLLL  AA    AA           VVV       IIIIII  EEEEEE
                                       v1.0b by Suzy
--------------------------------------------------
 Avec ce programme, vous pourrez jouer à Dieu ! Vous
 créerez la vie et verrez comment elle se comporte
 car vous ne pourrez ensuite que la regarder et
 constater la prolifération, le déclin ou le parfait
 équilibre de votre création...
--------------------------------------------------
                 MENU PRINCIPAL
--------------------------------------------------
 1 - Simulation rapide
 2 - Simulation avancée
 3 - A propos du Jeu de la Vie
 4 - Crédits
 5 - Quitter
 Votre choix : 1


--------------------------------------------------
          Paramètres de la simulation :
--------------------------------------------------
 Forme du modèle : tore
 Règle de vie : 3 cellules
 Règle de neutralité : 2 cellules
 Taille de la grille : 20x20
 Initialisation de la grille : Aléatoire
 Probabilité d'être vivant : 0.5

 Voulez-vous continuer ? o/n : o
```

Code : Console - Un tour

```
--------------------------------------------------
                  Tour n° : 6
--------------------------------------------------

 . . . . . O O O . O O . . . O . . O . .
 . . . . . . . . . . . . . . O . O . O .
 . . . . . . . . . O . O . . . . . O O .
 . . . . . . . . O . . O O . . . . . O .
 . . . . . . . . O . . O . . . . . . . .
 . . . . . . . O . . . O . . . . . . . .
 O . . . . . . . . O . . . . . . . O . O
 O O O . . O . . . . O O . O . O . O O O
 O O O O O O O . . O O O O . . . . . . O
 . . O O O . . . . O . . O O . O . . . .
 . . . . O . . O . . . . O O O . . . . .
 . . . . . . . . . . . . . O . . . . O . .
 . . . . . O . . . . . . . . . O O . O .
```

```
. . . . . O . O . . . O O . O . . . .
. . . . . . . . . . . . . O O O . . . .
. . . . . . . . O O . . O . . O . . . .
O . . . . . . . . . . . . . O . . . . . O
. . . . . . . . . . . . . O . . O . . . .
. . . . . . . . . . . . . O . . . . . O .
. . . . . . . . . . O O . . . O . O . . .

--------------------------------------------------
 1 - Continuer...
 2 - Avancer de plusieurs pas...
 3 - Sauvegarder et continuer...
 4 - Sauvegarder et avancer de plusieurs pas...
 5 - Sauvegarder et arrêter
 6 - Arrêter
 Votre choix :
```

J'ai essayé de faire une petite interface claire bien que ce soit un programme en console ! C'est toujours marrant d'avoir des programmes très bien présentés en console, le plus souvent, les interfaces en ASCII sont baclées et c'est bien dommage ! 

Corrigé

Ce corrigé ne sera certainement pas parfait et n'est que le reflet d'une manière de faire, la mienne. Ce n'est pas la meilleure manière de faire mais c'en est une que j'ai développée au fur et à mesure de mes années d'expérience de programmation notamment en Python. Ce corrigé est donc là uniquement pour donner une idée de comment il est possible de s'y prendre pour un tel projet et n'est en aucun cas un exemple à suivre à la lettre !

Organiser son code

Lorsque l'on veut commencer à programmer un programme assez complet, il faut commencer par s'organiser. Ici, je sépare le projet en deux grandes parties :

- Le programme principal qui affiche l'interface et qui interagit avec l'utilisateur.
- Le moteur de l'application qui va permettre de lancer les simulations.

Je vais donc créer un fichier/module pour chacune de ces parties. Je sépare ainsi l'interface du moteur de l'application. En général, cela permet de refaire une interface sans avoir à modifier son application lorsque celle-ci ne nous plaît pas ou si l'on souhaite développer une interface graphique plus tard par exemple. Le moteur pourra être réutilisé indépendant de l'interface par la suite. Ainsi, je pourrais améliorer mon programme sans jamais avoir à refaire un moteur de simulation !
J'ai donc créé les fichiers qu'il me faudra pour mon programme et je commence à organiser les fonctions qui me seront sûrement utiles. Cela me permet de prédécouper mon code et donc de mieux réfléchir à ce que je veux faire à petite échelle.

Moteur.py

Secret (cliquez pour afficher)

Code : Python

```
###################################
# Jeu de la vie #
# ------------------------------- #
# Auteur : Suzy #
# Créé le : 09/07/2010 #
# ------------------------------- #
# Ce petit programme de simulation #
# permet de modéliser une forme de #
# vie simplicime ! Vous pourrez y #
# voir apparaître des formes qui #
# se perpétuent dans le temps ou #
# qui restent constantes. #
# ------------------------------- #
# Moteur du jeu de la vie : #
# Contient toutes les fonctions #
# qui permettent d'effectuer des #
# opérations sur la grille et #
# d'émuler la vie ! #
###################################
```

```
from random import random

#Constantes :
FORMES_POSSIBLES = ("plan","cylindre","tore")

def creerGrille(lignes = 20, colonnes = 20, proba = 0.5) :
    """
Permet de créer une grille du nombre de lignes et de colonnes
spécifiées. Les cellules seront vivantes selon la probabilité
données en paramètre.
lignes = 20 : Le nombre de lignes pour la grille.
colonnes = 20 : La nombre de colonnes pour la grille.
proba = 0.5 : La probabilité qu'une cellule créée soit vivante.
Retourne : Retourne une grille de lignes x colonnes cellules
aléatoirement vivantes.
"""
    pass


def chargerGrille(nomFichier) :
    """
Permet de charger une grille à partir d'un fichier. Le fichier
doit être composé de lignes de 0 (mort) et de 1 (vivant). Si
elles ne sont pas toutes de la même taille, une exception sera
levée.
nomFichier : Le chemin vers le fichier à charger.
Retourne : La grille chargée à partir du fichier.
Exception : - IOError : Si le fichier n'existe pas.
- ValueError : quelque chose n'est pas un nombre dans la grille.
- AssertionError : un nombre différent de 0 ou 1.
- AssertionError : lignes de différentes longueurs.
- AssertionError : une seule ligne dans le fichier.
- AssertionError : pas de cellule vivante.
"""
    pass


def sauvegarderGrille(grille, nomFichier) :
    """
Permet d'enregistrer une grille dans un fichier qui pourra
être chargé par la suite. Le fichier sera préalablement effacé
s'il existe déjà.
grille : La grille à sauvegarder.
nomFichier : Le nom du fichier à utiliser pour la sauvegarde.
Retourne : 0 si l'opération s'est bien passée, 1 sinon.
"""
    pass


def voisines(grille, i, j, forme) :
    """
Permet d'obtenir une liste des voisines d'une cellule aux
coordonnées
demandées dans une grille demandée. La forme du modèle est prise
en compte.
grille : La grille contenant la cellule à étudier.
i : L'indice de la ligne de la cellule.
j : L'indice de la colonne de la cellule.
forme : La forme du modèle (plan, cylindre, tore).
Retourne : La liste des voisines de la cellule demandée.
Exception : - IndexError si i et/ou j sort de la grille.
- AssertionError si la forme demandée n'est pas reconnue.
"""
    pass


def tourSuivant(grille, regle1 = 3, regle2 = 2, forme = "tore") :
    """
Permet de créer le tour suivant de la grille en fonction des
```

```
    règles en vigueur dans la source de vie que vous avez créée.
    grille : La grille de cellules à faire évoluer.
    regle1 = 3 : Le nombre de cellules à avoir pour qu'une cellule
    soit vivante au prochain tour.
    regle2 = 2 : Le nombre de cellules à avoir pour qu'une cellule
    reste dans le même état au tour suivant.
    forme = "plan" : Le modèle peut prendre différentes formes, soit
    un plan fermé ("plan"), soit un cylindre ("cylindre"),
    soit un tore ("tore").
    Change les voisines aux bords du modèle.
    Retourne : Retourne la nouvelle grille après évolution.
    Exception : AssertionError : la forme donnée n'est pas reconnue.
    """
        pass


    def encoreDeLaVie(grille) :
        """
    Permet de savoir si la grille contient encore de la vie ou non.
    grille : La grille a tester.
    Retourne : Retourne True si la grille contient de la vie, False
    sinon.
    """
        pass
```

JeuDeLaVie.py

Secret (cliquez pour afficher)

Code : Python

```
    ###################################
    # Jeu de la vie                   #
    # ------------------------------- #
    # Auteur : Suzy                   #
    # Créé le : 09/07/2010            #
    # ------------------------------- #
    # Ce petit programme de simulation #
    # permet de modéliser une forme de #
    # vie simplicime ! Vous pourrez y #
    # voir apparaître des formes qui  #
    # se perpétuent dans le temps ou  #
    # qui restent constantes.         #
    # ------------------------------- #
    # Programme principal :           #
    # - Grille                        #
    # - Interface                     #
    # - Options                       #
    ###################################


    #Importation de mes modules perso :
    import Moteur
    #Importation des modules intégrés :
    import os, time, platform


    def nettoyerConsole() :
        """
    Permet de nettoyer la console !
    """
        pass


    def menuPrincipal() :
```

```
    """
Affiche le titre du programme avec un petit texte introductif.
Affiche ensuite le menu principal et récupère le choix de
l'utilisateur.
Lance ensuite la ou les simulations en fonction du choix...
"""
    pass


def simuRapide() :
    """
Lance une simu rapide de 20x20 en forme de tore ou propose de
revenir
en arrière.
"""
    pass


def simuAvancee() :
    """
Propose de définir tout un tas de paramètres pour lancer une
simulation
ou de revenir en arrière.
"""
    pass


def aPropos() :
    """
Affiche quelques infos à propos du jeu de la vie.
"""
    pass


def mesCredits() :
    """
Affiche les crédits.
"""
    pass


def lancerSimulation(grille, regle1 = 3, regle2 = 2, forme =
"tore") :
    """
Lance une simulation avec les règles et les options demandées.
grille : La grille initiale.
regle1 = 3 : La règle de vie (combien de cellules vivantes pour
naître)
regle2 = 2 : La règle de neutralité (combien de cellules vivantes
pour rester
inchangé.
forme = "tore" : La forme à considérer pour la grille.
"""
    pass


def lancerSimulationInfinie(grille, regle1, regle2, forme,
pasAvantSauvegarde, nomFichier) :
    """
Lance une simulation infinie qui ne sera stoppée que si le
modèle meurt ou si
l'utilisateur la tue !
grille : La grille initiale.
regle1 : La règle de vie (combien de cellules vivantes pour
naître)
regle2 : La règle de neutralité (combien de cellules vivantes
pour rester
inchangé.
forme : La forme à considérer pour la grille.
pasAvantSauvegarde : Le nombre de pas entre deux sauvegarde (0
```

```
    pour ne pas sauvegarder).
    nomFichier : Le nom du fichier de sauvegarde.
    """
        pass


    def afficherGrille(grille, tour) :
        """
    Convertit une grille en chaîne de caractères pour l'afficher
    dans la console.
    grille : La grille a afficher.
    tour : Le numéro de tour.
    """
        pass
```

Vous avez ici, une première ébauche du code qui composera le coeur de notre application. Pour m'organiser, j'ai commencé à créer les fonctions que j'aurais à utiliser pour pouvoir créer des simulations et faire évoluer tout ce petit monde dans le module Moteur.py et tout ce qui va permettre d'interagir avec l'utilisateur et de lancer des simulations dans le module JeuDeLaVie.py La documentation est bien entendu aussi complète que possible afin d'exprimer en langage naturel ce que l'on souhaite faire en code lors de la phase d'implémentation. Elle doit vous permettre de savoir à quoi celle-ci servira par la suite !

Pour ceux qui n'auraient pas d'idées sur comment faire l'interface, aidez-vous de ce que j'ai donné en exemple plus haut. Et surtout, ne cherchez pas à faire compliquer, vous pouvez vous contenter d'une petite interface basique !

Tester les fonctions

Une fois l'implémentation entamée, vous devrez tester votre code. Je vous conseille de toujours le tester au fur et à mesure afin de ne pas avoir à tout débugger d'un coup à la fin. Cela vous évitera de vous perdre dans les erreurs et vous pourrez ainsi les cibler beaucoup plus facilement si vous validez les fonctions les unes après les autres.

Dans notre cas, le module JeuDeLaVie.py sera le programme principal, il se contentera d'afficher la simulation et d'interagir avec l'utilisateur. Il ne nécessite pas de test particulier puisque vous verrez tout de suite le résultat en lançant votre programme ! Par contre, votre moteur lui, demande à être testé car certaines fonctions ne seront pas directement visibles de l'utilisateur. Pour cela, nous allons rajouter un code exécutable à la fin du module Moteur.py qui fera appel aux fonctions dans des cas bien précis. Ceci vous permettra de contrôler que le moteur de votre application se comporte comme vous l'espériez.

Secret (cliquez pour afficher)

Code : Python - A insérer dans Moteur.py

```
# Tests des fonctions :
if __name__ == "__main__" :
    print("********************\n" +
          "* Test du moteur ! *\n" +
          "********************")

    print("\n*****\n" +
          "* Création d'une grille vivante :\n" +
          "*****")
    grille = creerGrille(proba = 1)
    vivantes = 0
    for ligne in grille :
        print(ligne)
        for cellule in ligne :
            if cellule :
                vivantes += 1
    print("Nombre de cellules vivantes : %i (normalement à 400)" %
vivantes)

    print("\n*****\n" +
          "* Création d'une grille aléatoire :\n" +
          "*****")
    grille = creerGrille()
```

```
        vivantes = 0
        for ligne in grille :
            print(ligne)
            for cellule in ligne :
                if cellule :
                    vivantes += 1
        print("Nombre de cellules vivantes : %i (aux alentours de
    200)" % vivantes)

        print("\n*****\n" +
              "* Sauvegarde de la grille aléatoire :\n" +
              "*****")
        echec = sauvegarderGrille(grille, "Test/TestMoteur.txt")
        if echec :
            print("La grille n'a pas été sauvegardée...")
        else :
            print("Le fichier TestMoteur.txt a bien été écrit !")

        print("\n*****\n" +
              "* Chargement de la grille sauvegardée :\n" +
              "*****")
        grille2 = chargerGrille("Test/TestMoteur.txt")
        if grille2 == [] :
            print("Le fichier TestMoteur.txt n'a pas été chargé
    correctement...")
        else :
            print("Le fichier TestMoteur.txt a été correctement chargé
    !")
            for ligne in grille2 :
                print(ligne)

        print("\n*****\n" +
              "* Chargement d'une grille incorrecte :\n" +
              "*****")
        print("Lignes pas de la même longueur :")
        try :
            grille3 = chargerGrille("Test/Longueur.txt")
            print("La grille a tout de même été chargée...")
        except AssertionError:
            print("Le fichier existe mais est incorrect !")
        except :
            print("Le fichier ne semble pas exister...")
            raise
        print("Autre chose que 0 ou 1 dans le fichier :")

        try :
            grille4 = chargerGrille("Test/PasBonNombre.txt")
            print("La grille a tout de même été chargée...")
        except AssertionError:
            print("Le fichier existe mais est incorrect !")
        except :
            print("Le fichier ne semble pas exister...")
            raise

        print("\n\n**********************\n" +
              "* Test des voisines : *\n" +
              "**********************\n")
        grille = creerGrille()
        for ligne in grille :
            print(ligne)
        print("Test de la forme tore :")
        print("Test coin haut gauche :")
        print(voisines(grille, 0, 0, "tore"))
        print("Test coin haut droit :")
        print(voisines(grille, 0, 19, "tore"))
        print("Test coin bas gauche :")
        print(voisines(grille, 19, 0, "tore"))
        print("Test coin bas droit :")
        print(voisines(grille, 19, 19, "tore"))
        print("Test au milieu (4,4) :")
```

```
print(voisines(grille, 4, 4, "tore"))
print("Test côté haut (0, 4) :")
print(voisines(grille, 0, 4, "tore"))
print("Test côté bas (19, 4) :")
print(voisines(grille, 19, 4, "tore"))
print("Test côté gauche (4, 0) :")
print(voisines(grille, 4, 0, "tore"))
print("Test côté droit (4, 19) :")
print(voisines(grille, 4, 19, "tore"))

print("\nTest de la forme plan :")
print("Test coin haut gauche :")
print(voisines(grille, 0, 0, "plan"))
print("Test coin haut droit :")
print(voisines(grille, 0, 19, "plan"))
print("Test coin bas gauche :")
print(voisines(grille, 19, 0, "plan"))
print("Test coin bas droit :")
print(voisines(grille, 19, 19, "plan"))
print("Test au milieu (4,4) :")
print(voisines(grille, 4, 4, "plan"))
print("Test côté haut (0, 4) :")
print(voisines(grille, 0, 4, "plan"))
print("Test côté bas (19, 4) :")
print(voisines(grille, 19, 4, "plan"))
print("Test côté gauche (4, 0) :")
print(voisines(grille, 4, 0, "plan"))
print("Test côté droit (4, 19) :")
print(voisines(grille, 4, 19, "plan"))

print("\nTest de la forme cylindre :")
print("Test coin haut gauche :")
print(voisines(grille, 0, 0, "cylindre"))
print("Test coin haut droit :")
print(voisines(grille, 0, 19, "cylindre"))
print("Test coin bas gauche :")
print(voisines(grille, 19, 0, "cylindre"))
print("Test coin bas droit :")
print(voisines(grille, 19, 19, "cylindre"))
print("Test au milieu (4,4) :")
print(voisines(grille, 4, 4, "cylindre"))
print("Test côté haut (0, 4) :")
print(voisines(grille, 0, 4, "cylindre"))
print("Test côté bas (19, 4) :")
print(voisines(grille, 19, 4, "cylindre"))
print("Test côté gauche (4, 0) :")
print(voisines(grille, 4, 0, "cylindre"))
print("Test côté droit (4, 19) :")
print(voisines(grille, 4, 19, "cylindre"))

print("*****\n" +
      "* Test de quelques tours de simu :\n" +
      "*****")
grille = chargerGrille("Test/PeriodiqueStable.txt")
print("Grille initiale :")
for ligne in grille :
    print(ligne)
grille = tourSuivant(grille)
print("Grille après 1 tour :")
for ligne in grille :
    print(ligne)
grille = tourSuivant(grille)
print("Grille après 2 tours :")
for ligne in grille :
    print(ligne)
grille = tourSuivant(grille)
print("Grille après 3 tours :")
for ligne in grille :
    print(ligne)
```

Ce code va vous permettre de tester le bon fonctionnement des fonctions que vous aurez implémentées. Il n'est bien entendu en rien un test à 100% efficace mais il vous permettra de détecter la plupart des problèmes majeures et des erreurs que vous pourriez commettre dans le moteur de l'application.

Ce morceau de code est écrit en Python v3. Si vous désirez le passer en v2.7, comme d'habitude, il faudra faire attention aux fonctions *print* et *input*.

Ce code n'est en aucun cas destiné à être bien écrit ou à être optimisé. Ce n'est qu'un code de test qui ne sera pas exécuté lors du lancement de l'application ! Ne vous prenez pas la tête sur la qualité du code de test...

Implémenter

Si vous n'avez pas réussi jusqu'à maintenant à implémenter les fonctions que vous ai données, alors vous pouvez regarder cette partie pour vous aider. Essayez de faire par vous-même au moins une des deux parties, ne serait-ce qu'en utilisant le corrigé de celle que vous n'arrivez pas à implémenter...

JeuDeLaVie.py

Secret (cliquez pour afficher)

Code : Python

```
# -*- coding: utf-8 -*-
###################################
# Jeu de la vie #
# ------------------------------- #
# Auteur : Suzy #
# Créé le : 09/07/2010 #
# ------------------------------- #
# Ce petit programme de simulation #
# permet de modéliser une forme de #
# vie simplicime ! Vous pourrez y #
# voir apparaître des formes qui #
# se perpétuent dans le temps ou #
# qui restent constantes. #
# ------------------------------- #
# Programme principal : #
# - Grille #
# - Interface #
# - Options #
###################################


#Importation de mes modules perso :
import Moteur
#Importation des modules intégrés :
import os, time, platform


def nettoyerConsole() :
    """
Permet de nettoyer la console !
"""
    if platform.system() == "Windows" :
        os.system("cls")
    else :
        os.system("clear")


def menuPrincipal() :
    """
Affiche le titre du programme avec un petit texte introductif.
Affiche ensuite le menu principal et récupère le choix de
```

```
    l'utilisateur.
    Lance ensuite la ou les simulations en fonction du choix...
    """
        global continuerProgramme
        nettoyerConsole()
        print("\n\n" +
              " JJJJ EEEEEE UU UU DDDDD EEEEEE \n" +
              " JJJJ EEEEEE UU UU DDDDDD EEEEEE \n" +
              " JJ EE UU UU DD DD EE \n" +
              " JJ EE UU UU DD DD EE \n" +
              " JJ EEEE UU UU DD DD EEEE \n" +
              " JJJJ JJ EEEE UU UU DD DD EEEE \n" +
              " JJJJ JJ EE UU UU DD DD EE \n" +
              " JJ JJ EE UU UU DD DD EE \n" +
              " JJJJJJ EEEEEE UUUUUUUUU DDDDDD EEEEEE \n" +
              " JJJJ EEEEEE UUUUUUU DDDDD EEEEEE \n" +
              " \n" +
              " \n" +
              " LL AAAAAA VV VV IIIIII EEEEEE \n" +
              " LL AAAAAAAA VV VV IIIIII EEEEEE \n" +
              " LL AA AA VV VV II EE \n" +
              " LL AA AA VV VV II EE \n" +
              " LL AA AA VV VV II EEEE \n" +
              " LL AAAAAAAA VV VV II EEEE \n" +
              " LL AAAAAAAA VV VV II EE \n" +
              " LL AA AA VV VV II EE \n" +
              " LLLLLL AA AA VV VV IIIIII EEEEEE \n" +
              " LLLLLL AA AA VVV IIIIII EEEEEE \n" +
              " v1.0b by Suzy \n" +
              "-------------------------------------------------
-\n" +
              " Avec ce programme, vous pourrez jouer à Dieu ! Vous
\n" +
              " créerez la vie et verrez comment elle se comporte \n"
+
              " car vous ne pourrez ensuite que la regarder et \n" +
              " constater la prolifération, le déclin ou le parfait
\n" +
              " équilibre de votre création... \n" +
              "-------------------------------------------------
-\n" +
              " MENU PRINCIPAL \n" +
              "-------------------------------------------------
-\n" +
              " 1 - Simulation rapide \n" +
              " 2 - Simulation avancée \n" +
              " 3 - A propos du Jeu de la Vie \n" +
              " 4 - Crédits \n" +
              " 5 - Quitter ")

        choix = input(" Votre choix : ")
        while choix not in list("12345") :
            print(" Raté ! Essayez encore !")
            choix = input(" Votre choix : ")

        if choix == "1" :
            simuRapide()
        elif choix == "2" :
            simuAvancee()
        elif choix == "3" :
            aPropos()
        elif choix == "4" :
            mesCredits()
        elif choix == "5" :
            continuerProgramme = False


    def simuRapide() :
        """
    Lance une simu rapide de 20x20 en forme de tore ou propose de
```

```
        revenir
        en arrière.
        """
            print("\n\n" +
                  "-------------------------------------------------
        -\n" +
                  " Paramètres de la simulation : \n" +
                  "-------------------------------------------------
        -\n" +
                  " Forme du modèle : tore\n" +
                  " Règle de vie : 3 cellules\n" +
                  " Règle de neutralité : 2 cellules\n"
                  " Taille de la grille : 20x20\n" +
                  " Initialisation de la grille : Aléatoire\n"
                  " Probabilité d'être vivant : 0.5\n")

            valider = input(" Voulez-vous continuer ? o/n : ")
            while valider.upper() not in ["O", "OUI", "N", "NON"] :
                print(" Raté ! Essayez encore !")
                valider = input(" Voulez-vous continuer ? o/n : ")

            if valider.upper() in ["O", "OUI"] :
                lancerSimulation(Moteur.creerGrille())


        def simuAvancee() :
            """
        Propose de définir tout un tas de paramètres pour lancer une
        simulation
        ou de revenir en arrière.
        """
            print("\n\n" +
                  "-------------------------------------------------
        -\n" +
                  " Paramètres de la simulation : \n" +
                  "-------------------------------------------------")
            #Simulation infinie ou contrôlée :
            typeSimu = input(" Quel genre de simulation voulez-vous lancer
        ?\n" +
                             " 1 - Simulation contrôlée\n" +
                             " 2 - Simulation infinie\n" +
                             " Votre choix : ")
            while typeSimu not in list("12") :
                print(" Raté ! Essayez encore !")
                typeSimy = input(" Votre choix : ")
            if typeSimu == "2" :
                pasAvantSauvegarde = obtenirNombre(" Nb de pas entre deux
        sauvegardes\n (0 pour ne pas sauvegarder) : ")
                if pasAvantSauvegarde == 0 :
                    nomSauvegarde = "Pas de sauvegarde"
                else :
                    nomSauvegarde = input(" Nom du fichier de sauvegarde :
        ")
            #Charger une grille ou non :
            choixCreation = False
            while not choixCreation :
                creationGrille = input(" Vous préférez :\n" +
                                       " 1 - Générer une grille
        aléatoire\n" +
                                       " 2 - Charger une grille à partir
        d'un fichier\n" +
                                       " Votre choix : ")
                while creationGrille not in list("12") :
                    print(" Raté ! Essayez encore !")
                    creationGrille = input(" Votre choix : ")
                #Si une grille aléatoire doit être générée :
                if creationGrille == "1" :
                    lignes = obtenirNombre(" Nombre de lignes pour votre
        grille : ")
                    colonnes = obtenirNombre(" Nombre de colonnes pour
```

```
votre grille : ")
            proba = obtenirProba()
            choixCreation = True
        else :
            reponseCorrecte = False
            while not reponseCorrecte :
                nomFichier = input(" Nom du fichier à charger
(tapez \nannuler pour revenir en arrière) : ")
                while not os.path.exists(nomFichier) and not
nomFichier.upper() == "ANNULER":
                    print(" Ce fichier n'existe pas...")
                    nomFichier = input(" Nom du fichier à charger
(annuler pour annuler) : ")
                if nomFichier.upper() != "ANNULER" :
                    try :
                        Moteur.chargerGrille(nomFichier)
                        reponseCorrecte = True
                        choixCreation = True
                    except :
                        print("Le fichier n'est pas correct...")
                else :
                    reponseCorrecte = True
                    choixCreation = False
    #Forme du modèle :
    forme = input(" Forme du modèle :\n" +
                  " 1 - Plan fermé\n" +
                  " 2 - Cylindre\n" +
                  " 3 - Tore\n" +
                  " Votre choix : ")
    while forme not in list("123") :
        print(" Raté ! Essayez encore !")
        forme = input(" Votre choix : ")
    if forme == "1" :
        forme = "plan"
    elif forme == "2" :
        forme = "cylindre"
    elif forme == "3" :
        forme = "tore"
    #Règles :
    regle1 = obtenirNombre(" Règle de vie (par défaut : 3) : ")
    regle2 = obtenirNombre(" Règle de neutralité (par défaut : 2)
: ")
    while regle1 == regle2 :
        print("Les deux règles ne peuvent prendre en compte le
même nombre...")
        regle2 = obtenirNombre(" Règle de neutralité (par défaut :
2) : ")

    #Récapitulatif :
    print("\n\n" +
          "-------------------------------------------------
-\n" +
          " Paramètres de la simulation : \n" +
          "-------------------------------------------------")
    if typeSimu == "1" :
        print(" Type simulation : contrôlée")
    else :
        print(" Type simulation : infinie\n" +
              " Fichier de sauvegarde : %s" % nomSauvegarde)
    print(" Forme du modèle : %s\n" % forme +
          " Règle de vie : %i cellules\n" % regle1 +
          " Règle de neutralité : %i cellules" % regle2)
    if creationGrille == "1" :
        print(" Taille de la grille : %ix%i\n" % (lignes,
colonnes) +
              " Initialisation de la grille : Aléatoire\n" +
              " Probabilité d'être vivant : %f\n" % proba)
        grille = Moteur.creerGrille(lignes, colonnes, proba)
    else :
        grille = Moteur.chargerGrille(nomFichier)
```

```
        print(" Taille de la grille : %ix%i\n" % (len(grille),
len(grille[0])) +
              " Initialisation de la grille : Charger un
fichier\n" +
              " Nom du fichier à charger : %s\n" % nomFichier)

    #Continuer ou pas :
    valider = input(" Voulez-vous continuer ? o/n : ")
    while valider.upper() not in ["O", "OUI", "N", "NON"] :
        print(" Raté ! Essayez encore !")
        valider = input(" Voulez-vous continuer ? o/n : ")

    if valider.upper() in ["O", "OUI"] :
        if typeSimu == "1" :
            lancerSimulation(grille, regle1, regle2, forme)
        else :
            lancerSimulationInfinie(grille, regle1, regle2, forme,
pasAvantSauvegarde, nomSauvegarde)


def aPropos() :
    """
Affiche quelques infos à propos du jeu de la vie.
"""
    nettoyerConsole()
    print("\n\n" +
          "-------------------------------------------------
-\n" +
          " A PROPOS \n" +
          "-------------------------------------------------
-\n" +
          " Le jeu de la vie est un automate cellulaire crée en
\n" +
          " 1970 par John Horton Conway. Dans ce programme, la \n"
+
          " simulation se déroule en deux dimensions sur un \n" +
          " plan fermé, un tore ou un cylindre. \n" +
          " Par défaut, les règles appliquées pour la vie ou la
\n" +
          " mort des cellules sont les suivantes : \n" +
          " - Règle de vie : une cellule naît si elle est en- \n"
+
          " d'exactement 3 cellules vivantes. \n" +
          " - Règle de neutralité : une cellule conserve son \n" +
          " état si elle est entourée par exactement 2 cel- \n" +
          " les vivantes. \n" +
          " - Dans tous les autres cas, elle meurt. \n" +
          "-------------------------------------------------")
    input(" Appuyez sur entrée pour revenir au menu principal.")


def mesCredits() :
    """
Affiche les crédits.
"""
    nettoyerConsole()
    print("\n\n" +
          "-------------------------------------------------
-\n" +
          " CREDITS \n" +
          "-------------------------------------------------
-\n" +
          " Ce programme a été développé par Suzy alias Bastien
\n" +
          " Pietropaoli dans le cadre de la rédaction d'un tu- \n"
+
          " toriel sur le langage Python pour le Site du Zéro. \n"
+
          "\n" +
          " Ce programme est entièrement libre et toute partie \n"
```

```
        +
                  " peut être utilisée sans aucune contrainte. Un petit
    \n" +
                  " remerciement sera suffisant si une partie du code a
    \n" +
                  " été utilisée. \n" +
                  "-----------------------------------------------------")
        input(" Appuyez sur entrée pour revenir au menu principal.")


    def lancerSimulation(grille, regle1 = 3, regle2 = 2, forme =
    "tore") :
        """
    Lance une simulation avec les règles et les options demandées.
    grille : La grille initiale.
    regle1 = 3 : La règle de vie (combien de cellules vivantes pour
    naître)
    regle2 = 2 : La règle de neutralité (combien de cellules vivantes
    pour rester
    inchangé.
    forme = "tore" : La forme à considérer pour la grille.
    """
        #Variables :
        tour = 0
        continuer = True
        #Faire les tours :
        while continuer :
            nettoyerConsole()
            afficherGrille(grille,tour)
            #Tant qu'il y a de la vie...
            if Moteur.encoreDeLaVie(grille) :
                print(" 1 - Continuer...\n" +
                      " 2 - Avancer de plusieurs pas...\n" +
                      " 3 - Sauvegarder et continuer...\n" +
                      " 4 - Sauvegarder et avancer de plusieurs
    pas...\n" +
                      " 5 - Sauvegarder et arrêter\n" +
                      " 6 - Arrêter")
                choix = input(" Votre choix : ")
                while choix not in list("123456") :
                    print(" Raté ! Essayez encore !")
                    choix = input("Votre choix : ")
                #Une fois un choix correct obtenu :
                if choix == "1" :
                    grille = Moteur.tourSuivant(grille, regle1,
    regle2, forme)
                elif choix == "2" :
                    for i in range(obtenirNombre(" Nombre de pas à
    effectuer : ")) :
                        grille = Moteur.tourSuivant(grille, regle1,
    regle2, forme)
                        tour += 1
                        if not Moteur.encoreDeLaVie(grille) :
                            desolation(tour)
                            break
                    if Moteur.encoreDeLaVie(grille) :
                        tour -= 1
                elif choix == "3" :
                    sauvegarder(grille)
                    grille = Moteur.tourSuivant(grille, regle1,
    regle2, forme)
                elif choix == "4" :
                    sauvegarder(grille)
                    for i in range(obtenirNombre(" Nombre de pas à
    effectuer : ")) :
                        grille = Moteur.tourSuivant(grille, regle1,
    regle2, forme)
                        tour += 1
                        if not Moteur.encoreDeLaVie(grille) :
                            desolation(tour)
```

```
                        break
                    if Moteur.encoreDeLaVie(grille) :
                        tour -= 1
                elif choix == "5" :
                    sauvegarder(grille)
                    continuer = False
                elif choix == "6" :
                    continuer = False
            #Si la grille n'est que désolation...
            else :
                continuer = False
                desolation(tour)
            #Tour suivant :
            tour += 1


    def lancerSimulationInfinie(grille, regle1, regle2, forme,
    pasAvantSauvegarde, nomFichier) :
        """
    Lance une simulation infinie qui ne sera stoppée que si le
    modèle meurt ou si
    l'utilisateur la tue !
    grille : La grille initiale.
    regle1 : La règle de vie (combien de cellules vivantes pour
    naître)
    regle2 : La règle de neutralité (combien de cellules vivantes
    pour rester
    inchangé.
    forme : La forme à considérer pour la grille.
    pasAvantSauvegarde : Le nombre de pas entre deux sauvegarde (0
    pour ne pas sauvegarder).
    nomFichier : Le nom du fichier de sauvegarde.
    """
        tour = 0
        try :
            while Moteur.encoreDeLaVie(grille) :
                nettoyerConsole()
                afficherGrille(grille,tour)
                print(" Pressez Ctrl+C pour arrêter la simulation.")
                grillePrecedente = grille
                grille = Moteur.tourSuivant(grille, regle1, regle2,
    forme)
                tour += 1
                #Attendre un peu :
                time.sleep(0.2)
                if pasAvantSauvegarde != 0 and tour%pasAvantSauvegarde
    == 0 and tour != 0 :
                    Moteur.sauvegarderGrille(grille, nomFichier)
                #Si la grille n'évolue plus :
                if grille == grillePrecedente :
                    stabilite()
                    assert False
            #Si l'on sort, c'est que la grille est morte :
            nettoyerConsole()
            afficherGrille(grille, tour)
            desolation(tour)
        except :
            nettoyerConsole()
            afficherGrille(grille,tour)
            print(" La simulation a été interrompue : \n" +
                  " 1 - Sauvegarder et revenir au menu principal \n" +
                  " 2 - Revenir au menu principal ")
            choix = input(" Votre choix : ")
            while choix not in list("12") :
                print(" Raté ! Essayez encore !")
                choix = input(" Votre choix : ")

            if choix == "1" :
                sauvegarder(grille)
```

```
def desolation(tour) :
    """
Affiche la fin de la simulation suite à la mort du modèle !
tour : Tour de la mort.
"""
    print(" DESOLATION ET DESESPOIR ! \n" +
          "-------------------------------------------------
-\n" +
          " Malheureusement, la vie que vous avez créé s'est \n" +
          " éteinte après %i tours... Vous ne semblez pas être \n"
% tour +
          " très doué... Mais vous pouvez retenter votre
chance.\n" +
          "-------------------------------------------------------")
    input(" Appuyez sur entrée pour continuer.")


def stabilite() :
    """
Affiche la fin de la simulation suite à sa stabilité !
tour : Tour de la stabilité.
"""
    print(" VIE ETERNELLE ! \n" +
          "-------------------------------------------------
-\n" +
          " La vie que vous avez créé a atteint la stabilité \n" +
          " parfaite ! Elle continuera son chemin jusqu'à l'in-
\n" +
          " fini et au-delà. Elle n'a plus besoin de gardien ! \n"
+
          "-------------------------------------------------------")
    input(" Appuyez sur entrée pour continuer.")


def sauvegarder(grille) :
    """
Permet de sauvegarder la grille.
grille : La grille à sauvegarder.
"""
    nomFichier = input(" Nom du fichier de sauvegarde : ")
    echec = Moteur.sauvegarderGrille(grille, nomFichier)
    while echec :
        print(" La sauvegarde n'a pas pu se faire...")
        nomFichier = input(" Nom du fichier de sauvegarde : ")
        echec = Moteur.sauvegarderGrille(grille, nomFichier)


def obtenirNombre(aEcrire) :
    """
Permet d'obtenir un nombre entier de l'utilisateur.
aEcrire : La phrase à écrire pour demander le nombre.
Retourne : Le nombre demandé.
"""
    reponseEstUnNombre = False
    while not reponseEstUnNombre :
        nombre = input(aEcrire)
        try :
            nombre = int(nombre)
            reponseEstUnNombre = True
        except :
            print(" Vous devriez donner un nombre...")
    return nombre


def obtenirProba() :
    """
Permet d'obtenir un nombre à virgule flottante de l'utilisateur
compris entre 0 et 1.
Retourne : Le nombre demandé.
```

```
    """
        reponseEstUnNombre = False
        while not reponseEstUnNombre :
            nombre = input(" Probabilité pour une cellule d'être
vivante au départ : ")
            try :
                nombre = float(nombre)
                if 0 <= nombre <= 1 :
                    reponseEstUnNombre = True
                else :
                    print(" La probabilité doit être comprise entre 0
et 1...")
            except :
                print(" Vous devriez donner un nombre...")
        return nombre


    def afficherGrille(grille, tour) :
        """
    Convertit une grille en chaîne de caractères pour l'afficher
    dans la console.
    grille : La grille a afficher.
    tour : Le numéro de tour.
    """
        aEcrire = "\n\n" + \
                  "-------------------------------------------------
--\n" + \
                  " Tour n° : %i \n" % tour + \
                  "-------------------------------------------------
--\n\n "
        for ligne in grille :
            for cellule in ligne :
                if cellule :
                    aEcrire += "O "
                else :
                    aEcrire += ". "
            aEcrire += "\n "
        aEcrire += "\n-------------------------------------------------
-----"
        print(aEcrire)


    # Programme principal :
    if __name__ == "__main__" :
        continuerProgramme = True
        while continuerProgramme :
            menuPrincipal()
```

Moteur.py

Secret (cliquez pour afficher)

Code : Python

```
    # -*- coding: utf-8 -*-
    ###################################
    # Jeu de la vie #
    # ------------------------------ #
    # Auteur : Suzy #
    # Créé le : 09/07/2010 #
    # ------------------------------ #
    # Ce petit programme de simulation #
    # permet de modéliser une forme de #
    # vie simplicime ! Vous pourrez y #
    # voir apparaître des formes qui #
    # se perpétuent dans le temps ou #
    # qui restent constantes. #
```

```
# ------------------------------ #
# Moteur du jeu de la vie : #
# Contient toutes les fonctions #
# qui permettent d'effectuer des #
# opérations sur la grille et #
# d'émuler la vie ! #
#####################################

from random import random

#Constantes :
FORMES_POSSIBLES = ("plan","cylindre","tore")

def creerGrille(lignes = 20, colonnes = 20, proba = 0.5) :
    """
Permet de créer une grille du nombre de lignes et de colonnes
spécifiées. Les cellules seront vivantes selon la probabilité
données en paramètre.
lignes = 20 : Le nombre de lignes pour la grille.
colonnes = 20 : La nombre de colonnes pour la grille.
proba = 0.5 : La probabilité qu'une cellule créée soit vivante.
Retourne : Retourne une grille de lignes x colonnes cellules
aléatoirement vivantes.
"""
    aRetourner = []
    for i in range(lignes) :
        ligne = []
        for j in range(colonnes) :
            if random() > proba :
                ligne.append(0)
            else :
                ligne.append(1)
        aRetourner.append(ligne)
    return aRetourner


def chargerGrille(nomFichier) :
    """
Permet de charger une grille à partir d'un fichier. Le fichier
doit être composé de lignes de 0 (mort) et de 1 (vivant). Si
elles ne sont pas toutes de la même taille, une exception sera
levée.
nomFichier : Le chemin vers le fichier à charger.
Retourne : La grille chargée à partir du fichier.
Exception : - IOError : Si le fichier n'existe pas.
- ValueError : quelque chose n'est pas un nombre dans la grille.
- AssertionError : un nombre différent de 0 ou 1.
- AssertionError : lignes de différentes longueurs.
- AssertionError : une seule ligne dans le fichier.
- AssertionError : pas de cellule vivante.
"""
    aRetourner = []
    #Lecture du fichier :
    fichier = open(nomFichier, 'r')
    for ligne in fichier :
        ligne = ligne.split()
        #Conversion en entier :
        for i, cellule in enumerate(ligne) :
            ligne[i] = int(cellule)
            #Vérification 0 ou 1 :
            assert ligne[i] == 0 or ligne[i] == 1
        aRetourner.append(ligne)
    fichier.close()
    #Vérification de la longueur des lignes :
    memeLongueur = True
    longueur = len(aRetourner[0])
    for ligne in aRetourner :
        memeLongueur = memeLongueur and len(ligne) == longueur
    assert memeLongueur
    assert len(aRetourner) > 1
```

```
        assert encoreDeLaVie(aRetourner)
        return aRetourner


    def sauvegarderGrille(grille, nomFichier) :
        """
    Permet d'enregistrer une grille dans un fichier qui pourra
    être chargé par la suite. Le fichier sera préalablement effacé
    s'il existe déjà.
    grille : La grille à sauvegarder.
    nomFichier : Le nom du fichier à utiliser pour la sauvegarde.
    Retourne : 0 si l'opération s'est bien passée, 1 sinon.
    """
        aEcrire = ""
        for ligne in grille :
            for cellule in ligne :
                aEcrire += "%i " % cellule
            aEcrire += "\n"
        try :
            fichier = open(nomFichier, "w")
            fichier.write(aEcrire)
            fichier.close()
            return 0
        except :
            return 1


    def voisines(grille, i, j, forme) :
        """
    Permet d'obtenir une liste des voisines d'une cellule aux
    coordonnées
    demandées dans une grille demandée. La forme du modèle est prise
    en compte.
    grille : La grille contenant la cellule à étudier.
    i : L'indice de la ligne de la cellule.
    j : L'indice de la colonne de la cellule.
    forme : La forme du modèle (plan, cylindre, tore).
    Retourne : La liste des voisines de la cellule demandée.
    Exception : - IndexError si i et/ou j sort de la grille.
    - AssertionError si la forme demandée n'est pas reconnue.
    """
        aRetourner = []
        nbLignes = len(grille)
        nbColonnes = len(grille[0])
        if forme == "plan" :
            #On considère 9 cellules :
            for k in range(-1, 2) :
                for l in range(-1, 2) :
                    if 0 <= (i + k) < nbLignes and 0 <= (j + l) <
    nbColonnes :
                        aRetourner.append(grille[i + k][j + l])
            #On retire la cellule elle-même (ou une valeur
    équivalente...) :
            aRetourner.remove(grille[i][j])
        elif forme == "cylindre" :
            #On considère 9 cellules :
            for k in range(-1, 2) :
                for l in range(-1, 2) :
                    if 0 <= (i + k) < nbLignes :
                        aRetourner.append(grille[i + k][(j +
    l)%nbColonnes])
            #On retire la cellule elle-même (ou une valeur
    équivalente...) :
            aRetourner.remove(grille[i][j])
        elif forme == "tore" :
            #On considère 9 cellules :
            for k in range(-1, 2) :
                for l in range(-1, 2) :
                    aRetourner.append(grille[(i + k)%nbLignes][(j +
    l)%nbColonnes])
```

```
            #On retire la cellule elle-même (ou une valeur
        équivalente...) :
            aRetourner.remove(grille[i][j])
        else :
            assert False
        return aRetourner


    def tourSuivant(grille, regle1 = 3, regle2 = 2, forme = "tore") :
        """
    Permet de créer le tour suivant de la grille en fonction des
    règles en vigueur dans la source de vie que vous avez créée.
    grille : La grille de cellules à faire évoluer.
    regle1 = 3 : Le nombre de cellules à avoir pour qu'une cellule
    soit vivante au prochain tour.
    regle2 = 2 : Le nombre de cellules à avoir pour qu'une cellule
    reste dans le même état au tour suivant.
    forme = "plan" : Le modèle peut prendre différentes formes, soit
    un plan fermé ("plan"), soit un cylindre ("cylindre"),
    soit un tore ("tore").
    Change les voisines aux bords du modèle.
    Retourne : Retourne la nouvelle grille après évolution.
    Exception : AssertionError : la forme donnée n'est pas reconnue.
    """
        global FORMES_POSSIBLES
        assert forme in FORMES_POSSIBLES
        aRetourner = []
        #Parcours de la grille :
        for i, ligne in enumerate(grille) :
            nouvelleLigne = []
            for j, cellule in enumerate(ligne) :
                #Compter les voisines vivantes :
                voisinesVivantes = sum(voisines(grille, i, j, forme))
                #Appliquer les règles de vie ou de mort :
                if voisinesVivantes == regle1 :
                    nouvelleLigne.append(1)
                elif voisinesVivantes == regle2 :
                    nouvelleLigne.append(cellule)
                else :
                    nouvelleLigne.append(0)
            aRetourner.append(nouvelleLigne)
        return aRetourner


    def encoreDeLaVie(grille) :
        """
    Permet de savoir si la grille contient encore de la vie ou non.
    grille : La grille a tester.
    Retourne : Retourne True si la grille contient de la vie, False
    sinon.
    """
        aRetourner = False
        for ligne in grille :
            for cellule in ligne :
                aRetourner = aRetourner or cellule
        return aRetourner


    # Tests des fonctions :
    if __name__ == "__main__" :
        print("********************\n" +
              "* Test du moteur ! *\n" +
              "********************")

        print("\n*****\n" +
              "* Création d'une grille vivante :\n" +
              "*****")
        grille = creerGrille(proba = 1)
        vivantes = 0
        for ligne in grille :
```

```
            print(ligne)
            for cellule in ligne :
                if cellule :
                    vivantes += 1
        print("Nombre de cellules vivantes : %i (normalement à 400)" %
    vivantes)

        print("\n*****\n" +
              "* Création d'une grille aléatoire :\n" +
              "*****")
        grille = creerGrille()
        vivantes = 0
        for ligne in grille :
            print(ligne)
            for cellule in ligne :
                if cellule :
                    vivantes += 1
        print("Nombre de cellules vivantes : %i (aux alentours de
    200)" % vivantes)

        print("\n*****\n" +
              "* Sauvegarde de la grille aléatoire :\n" +
              "*****")
        echec = sauvegarderGrille(grille, "Test/TestMoteur.txt")
        if echec :
            print("La grille n'a pas été sauvegardée...")
        else :
            print("Le fichier TestMoteur.txt a bien été écrit !")

        print("\n*****\n" +
              "* Chargement de la grille sauvegardée :\n" +
              "*****")
        grille2 = chargerGrille("Test/TestMoteur.txt")
        if grille2 == [] :
            print("Le fichier TestMoteur.txt n'a pas été chargé
    correctement...")
        else :
            print("Le fichier TestMoteur.txt a été correctement chargé
    !")
            for ligne in grille2 :
                print(ligne)

        print("\n*****\n" +
              "* Chargement d'une grille incorrecte :\n" +
              "*****")
        print("Lignes pas de la même longueur :")
        try :
            grille3 = chargerGrille("Test/Longueur.txt")
            print("La grille a tout de même été chargée...")
        except AssertionError:
            print("Le fichier existe mais est incorrect !")
        except :
            print("Le fichier ne semble pas exister...")
            raise
        print("Autre chose que 0 ou 1 dans le fichier :")

        try :
            grille4 = chargerGrille("Test/PasBonNombre.txt")
            print("La grille a tout de même été chargée...")
        except AssertionError:
            print("Le fichier existe mais est incorrect !")
        except :
            print("Le fichier ne semble pas exister...")
            raise

        print("\n\n**********************\n" +
              "* Test des voisines : *\n" +
              "**********************\n")
        grille = creerGrille()
        for ligne in grille :
```

```
        print(ligne)
    print("Test de la forme tore :")
    print("Test coin haut gauche :")
    print(voisines(grille, 0, 0, "tore"))
    print("Test coin haut droit :")
    print(voisines(grille, 0, 19, "tore"))
    print("Test coin bas gauche :")
    print(voisines(grille, 19, 0, "tore"))
    print("Test coin bas droit :")
    print(voisines(grille, 19, 19, "tore"))
    print("Test au milieu (4,4) :")
    print(voisines(grille, 4, 4, "tore"))
    print("Test côté haut (0, 4) :")
    print(voisines(grille, 0, 4, "tore"))
    print("Test côté bas (19, 4) :")
    print(voisines(grille, 19, 4, "tore"))
    print("Test côté gauche (4, 0) :")
    print(voisines(grille, 4, 0, "tore"))
    print("Test côté droit (4, 19) :")
    print(voisines(grille, 4, 19, "tore"))

    print("\nTest de la forme plan :")
    print("Test coin haut gauche :")
    print(voisines(grille, 0, 0, "plan"))
    print("Test coin haut droit :")
    print(voisines(grille, 0, 19, "plan"))
    print("Test coin bas gauche :")
    print(voisines(grille, 19, 0, "plan"))
    print("Test coin bas droit :")
    print(voisines(grille, 19, 19, "plan"))
    print("Test au milieu (4,4) :")
    print(voisines(grille, 4, 4, "plan"))
    print("Test côté haut (0, 4) :")
    print(voisines(grille, 0, 4, "plan"))
    print("Test côté bas (19, 4) :")
    print(voisines(grille, 19, 4, "plan"))
    print("Test côté gauche (4, 0) :")
    print(voisines(grille, 4, 0, "plan"))
    print("Test côté droit (4, 19) :")
    print(voisines(grille, 4, 19, "plan"))

    print("\nTest de la forme cylindre :")
    print("Test coin haut gauche :")
    print(voisines(grille, 0, 0, "cylindre"))
    print("Test coin haut droit :")
    print(voisines(grille, 0, 19, "cylindre"))
    print("Test coin bas gauche :")
    print(voisines(grille, 19, 0, "cylindre"))
    print("Test coin bas droit :")
    print(voisines(grille, 19, 19, "cylindre"))
    print("Test au milieu (4,4) :")
    print(voisines(grille, 4, 4, "cylindre"))
    print("Test côté haut (0, 4) :")
    print(voisines(grille, 0, 4, "cylindre"))
    print("Test côté bas (19, 4) :")
    print(voisines(grille, 19, 4, "cylindre"))
    print("Test côté gauche (4, 0) :")
    print(voisines(grille, 4, 0, "cylindre"))
    print("Test côté droit (4, 19) :")
    print(voisines(grille, 4, 19, "cylindre"))

    print("*****\n" +
          "* Test de quelques tours de simu :\n" +
          "*****")
    grille = chargerGrille("Test/PeriodiqueStable.txt")
    print("Grille initiale :")
    for ligne in grille :
        print(ligne)
    grille = tourSuivant(grille)
    print("Grille après 1 tour :")
```

```
        for ligne in grille :
            print(ligne)
        grille = tourSuivant(grille)
        print("Grille après 2 tours :")
        for ligne in grille :
            print(ligne)
        grille = tourSuivant(grille)
        print("Grille après 3 tours :")
        for ligne in grille :
            print(ligne)
        input()
```

Dans mon modèle, j'ai décidé de représenter les cellules par un booléen sous forme d'entier. Cette forme à l'avantage de permettre de tester rapidement si une cellule est morte ou vivante mais cela permet aussi de compter facilement les cellules vivantes autour d'une cellule à tester par exemple. J'ai donc essayé de simplifier un peu l'écriture des algorithmes en prenant un modèle qui s'adapte assez bien au problème que l'on traite ici !

Vous pourrez trouver ma version complète accompagné de quelques fichiers de tests et quelques configurations initiales sympathiques en cliquant ici.
Ce TP est un bon exercice si vous avez réussi à le faire en entier mais vous ne pourrez apprendre à programmer qu'en vous exerçant encore et encore ! Essayer donc de trouver des petits projets que vous pourriez faire. Les sites d'école informatique en proposent assez souvent et sinon, vous pouvez vous essayer aux grands classiques des jeux !

Vous voilà prêt pour passer à la programmation par objet. Avant d'y passer, assurez-vous de bien maîtriser les fonctions, car vous en aurez besoin à tour de bras !

Partie 3 : Programmation Orientée Objet

Ca fait peur comme nom hein ? En fait, vous allez voir qu'une fois que vous y aurez goûté, vous ne pourrez plus vous en passer ! La programmation orientée objet, c'est le must de l'intuitif pour la structuration d'un code et ça permet de simplifier et de rendre réutilisables des pans entiers de code ! Alors, ne perdons pas de temps et commençons !

Un objet, c'est quoi ?

La programmation orientée objet est avant tout composée de concepts. Nous allons donc voir dans cette partie, les concepts de base que vous devez vous approprier afin de devenir des programmeurs objets ! 

Dans cette partie, nous aborderons alors :

- Les objets que vous utilisez déjà
- Les concepts que l'on y retrouve
- Beaucoup de vocabulaire

Au niveau pratique, vous n'apprendrez rien de nouveau mais tout ce qui se trouve dans cette partie doit être connu sur le bout des doigts pour comprendre la suite ! La pratique ne sera pas possible sans un peu de théorie !

Mmh, vous avez dit objet ?

Commençons ce chapitre par une petite définition de la programmation objet :

Citation : Wikipédia

> C'est un paradigme de programmation informatique qui consiste en la définition et l'interaction de briques logicielles appelées objets ; un objet représente un concept, une idée ou toute entité du monde physique, comme une voiture, une personne ou encore une page d'un livre. Il possède une structure interne et un comportement, et il sait communiquer avec ses pairs. Il s'agit donc de représenter ces objets et leurs relations ; la communication entre les objets via leur relation permet de réaliser les fonctionnalités attendues, de résoudre le ou les problèmes.

Pour un zéro, il y a de fortes chances pour que cette définition ne soit pas très claire. Je vais donc essayer de l'expliciter. En gros, dans la programmation orientée objet, on crée des objets comme dans le monde physique. Chaque objet est représenté par des caractéristiques et des actions que l'on peut lui faire subir. Par exemple, dans la vraie vie, vous pouvez prendre un objet livre composé de pages que vous pouvez tourner.

En programmation, des termes particuliers sont utilisés pour décrire les caractéristiques et les actions que l'on peut faire sur un objet :

- **Les attributs** : c'est ce qui définit un objet (pour le livre, ses dimensions, son nombre de page, son titre, etc...)
- **Les méthodes** : ce sont les actions que l'on peut faire avec cet objet (pour le livre, ouvrir/fermer, tourner les pages, lire, etc...)

Le fait de modéliser des objets physiques que l'on peut appréhender rend la programmation orientée objet relativement instinctive. En effet, dans la plupart des problèmes que vous chercherez à résoudre, vous en aurez une représentation naturelle avec des objets qui vous viendra à l'esprit puisque vous réfléchirez d'abord à comment vous feriez en réalité pour faire ce que vous voulez programmer. Vous verrez ensuite que la plupart des algorithmes seront faciles à penser une fois votre structure bien définie.

D'ailleurs, si vous pensiez n'avoir jamais fait de POO, et bien c'est faux ! Jusqu'à maintenant vous avez déjà manipulé des objets. En effet, les listes, les dictionnaires, les fichiers et toutes les séquences en général sont des objets que vous manipulez sans problème !

Code : Python

```
liste = []
liste.append(12)
liste.append("Peuh !")
```

Dans ce petit exemple, on crée un objet de type *list* et on lui applique la méthode *append*(). Remarquez ici que *list* est bien un type de donnée ! En fait, lorsque vous créez des objets, vous créez de nouveaux types de données ! Pour appliquer une méthode à un objet en particulier, vous l'appliquez à l'aide du point (notation pointée). Vous pouvez retrouver la même chose avec les

fichiers...

Code : Python

```
fichier = open("FichierQuelconque.txt","w")
fichier.write("Peuh !\nJe suis un code de test !\nKTHXBYE")
fichier.close()
```

Dans ce cas, un fichier est ouvert, la méthode *write()* appliquée à ce fichier permet d'y écrire du texte et la méthode *close()* permet de le fermer. Voyons encore quelques points de théorie avant de créer nos propres objets.

Principe d'encapsulation

En programmation orientée objet, il existe un concept très important à respecter le plus possible pour s'éviter toute sorte de problèmes, c'est le principe d'encapsulation des données. Mais qu'est-ce que c'est ? Wikipédia nous dit :

Citation : Wikipédia

> En programmation orientée objet, l'encapsulation est l'idée de protéger l'information contenue dans un objet et de ne proposer que des méthodes de manipulation de cet objet. Ainsi, les propriétés et axiomes associés aux informations contenues dans l'objet seront assurés/validés par les méthodes de l'objet et ne seront plus de la responsabilité de l'utilisateur extérieur. L'utilisateur extérieur ne pourra pas modifier directement l'information et risquer de mettre en péril les axiomes et les propriétés comportementales de l'objet.

L'encapsulation des données consiste donc à ne pas mettre au jour les données caractéristiques d'un objet. Ainsi, les attributs d'un objet ne doivent pas être manipulés directement par l'utilisateur. En clair, lorsque vous utilisez un objet, vous utilisez une boîte noire munie d'une interface.

Mais alors, ça sert à quoi de créer des objets, d'y stocker des données si je ne peux pas y accéder ?!

Et bien, les méthodes qui peuvent être appliquées à l'objet sont l'interface entre un modèle (attributs et axiomes) et l'utilisateur final, vous. Les données ne pourront être manipulées que via des méthodes. Les méthodes qui permettent d'accéder ou de modifier les attributs sont respectivement appelées **accesseurs** et **mutateurs**. Vous verrez dans la suite que ce sont des méthodes différenciées uniquement par des conventions.

L'exemple le plus utilisé pour illustrer l'encapsulation est celui des nombres complexes. Ce sont des nombres que l'on peut écrire de façon cartésienne ou polaire. Dans un cas comme dans l'autre, on peut en obtenir des informations comme la partie réelle, la partie imaginaire, le module, l'argument, etc... Seulement, suivant s'il est stocké en mémoire sous forme cartésienne ou polaire, ces données ne s'obtiennent pas de la même façon. Mais finalement, est-ce que cela intéresse l'utilisateur final ? Et bien non, lui, il n'a besoin que de savoir qu'il peut récupérer ces informations quoiqu'il arrive ! Le modèle est donc pour lui transparent. Il ne le perçoit pas.

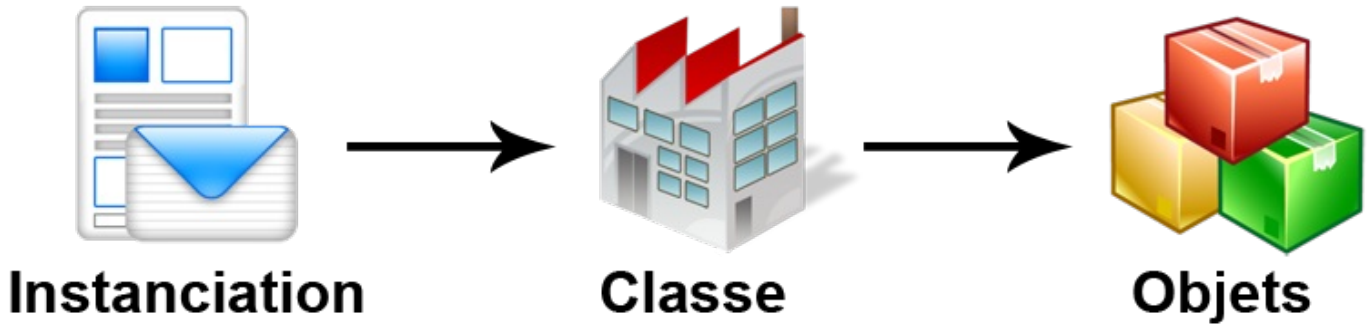
Pour ceux d'entre vous qui n'ont pas forcément compris l'exemple des nombres complexes, vous pouvez considérer votre ordinateur. Le modèle, c'est ce qu'il y a dans votre ordinateur, les opérations que peut effectuer votre processeur tout ça. L'interface, c'est le clavier, la souris et l'écran. Et vous pouvez appliquer des méthodes à votre ordinateur à l'aide d'applications. Avez-vous besoin de savoir comment fonctionnent les applications et votre ordinateur pour les utiliser ? Dans la plupart des cas, pas du tout ! Le modèle est totalement transparent une fois de plus ! Votre matériel respecte donc le principe d'encapsulation !

En conclusion, il faut toujours cacher votre modèle à l'utilisateur ! Nous verrons dans le chapitre de pratique comment le faire.

C'est la classe !

Ce que nous avons vu jusqu'à maintenant en manipulant les listes, les dictionnaires, etc... c'est qu'il est possible de créer plusieurs objets d'un même type et qu'il est aussi possible de définir de nouveaux types de données. Pour faire cela, il faut donc une sorte d'usine à objet : **une classe**. Une classe est ce qui sera défini pour la création d'objets. Les objets ainsi créés sont appelés **instances**.

Pour les fans de MMORPG (prononcé Meuporg selon certains journalistes...), vous comprenez certainement mieux le nom d'instance qui est donné aux donjons dont une nouvelle instance est créée pour chaque groupe de joueurs !



Sur ce schéma, vous passez votre commande (instanciation/création d'instance), et l'usine (la classe) vous fabrique les objets que vous avez demandé avec les caractéristiques demandées. Tous les objets issus d'une même classe ont tous les mêmes attributs mais ces attributs n'auront pas forcément les mêmes valeurs d'une instance à une autre ! Toutes les boîtes sont des boîtes mais n'ont pas la même couleur par exemple. La classe est donc le nouveau type de donnée que vous désirez créer. Elle vous permettra de créer autant d'instances que vous le désirerez. Dans la partie pratique, nous verrons donc comment définir tout ce que nous avons vu dans ce chapitre :

- **Des classes** : pour créer de nouveaux types de donnée
- **Les attributs** : les différents types et comment les définir
- **Les méthodes** : les différents types et comment les définir
- **Les accesseurs et les mutateurs** : les créer en respectant les conventions
- **Le constructeur** : pour créer des instances
- **Les instances** : comment les créer et les manipuler

Certains de ces termes ne seront définis que dans la suite et tous seront explicités avec des exemples de code. Ils sont à connaître car ce sont les termes techniques dont vous aurez besoin pour comprendre la suite du cours !
Bien, maintenant que vous avez une petite idée de ce qu'est la programmation orientée objet, nous allons voir ce que cela donne en pratique avec la création de vos propres objets ! Parce que c'est bien joli de faire de la théorie, mais si on n'en fait rien, je n'en vois pas l'intérêt ! 

Créer des objets (1/2)

Après quelques définitions utiles à la compréhension de la programmation orientée objet, et ce indépendamment du langage de programmation, nous allons passer à la pratique pour le langage Python. Dans ce chapitre, nous allons apprendre à

- Définir une classe
- Implémenter le constructeur et manipuler les attributs
- Implémenter et utiliser des méthodes d'instance

Le but de ce chapitre est d'être capable de créer de petites classes de base puis de les utiliser. Rien qu'avec ce chapitre, vous devriez être capable de comprendre un peu le fonctionnement des objets que vous avez utilisés jusqu'à maintenant.

Définir une classe

Alors, si vous avez bien suivi le chapitre précédent, vous devez savoir qu'une classe est une sorte d'usine à objets. Vous pourrez donc créer des instances de classe par la suite et ce autant de fois que vous le désirez !

Pour déclarer une classe, il faut utiliser le mot clé *class*. Difficile hein ? 

Code : Python

```
class MaPremiereClasse :
    """
Je prends toujours le temps de documenter mes classes
pour que tout le monde sache à quoi elles servent !
C'est un bon réflexe à prendre car je peux revenir sur
mon code des mois après et le comprendre en quelques
minutes.
"""
    pass
```

La définition d'une classe est donc une chose très simple. Il suffit d'utiliser le mot *class* puis de le faire suivre du nom de votre classe. Comme pour les fonctions, il est possible de documenter ses classes en faisant suivre la déclaration immédiatement d'un texte entourée de 'triple quote'.

Les noms de classe commencent généralement par une majuscule et sont composés de noms car ils représentent des objets. Ainsi, on peut les différencier facilement des noms de fonction qui commencent par une minuscule et qui auront plutôt tendance à contenir des verbes d'action.

En Python, comme pour le reste, une classe va être considérée comme une structure et l'indentation des blocs est donc ce qui permet de déterminer ce qui s'y trouve ! Il faudra alors indenter tout le code que vous désirez insérer dans une classe.

D'ailleurs, notre classe ne sert pour le moment à rien du tout ! Nous allons donc passer à la suite immédiatement.

Constructeur et attributs

Avant de commencer à utiliser nos propres objets, il faut déjà que nous les définissions et nous leur donnions des attributs. C'est ce que nous allons voir tout de suite !

Constructeur

Pour créer des instances de votre classe, vous aurez besoin d'un constructeur. C'est une méthode spéciale qui permet, comme son nom l'indique, de construire des objets. Cette fonction va en fait initialiser l'objet que vous créerez. Nous en avons déjà vu. En fait, lorsque vous utilisiez les fonctions *list()*, *dict()*, etc... vous utilisiez déjà l'équivalent d'un constructeur. Ces fonctions créent et renvoient un objet du type voulu et contenant ce que vous lui passez en argument. Et bien lorsque vous implémentez un constructeur, c'est pareil.

En Python, le constructeur porte toujours le même nom : *__init__()*. C'est donc cette fonction qu'il va falloir implémenter pour pouvoir personnaliser vos objets.

Code : Python

```
class MaPremiereClasse :
    """
Documentation
"""
```

```
    def __init__(self) :
        """
Constructeur.
"""
        pass
```

Il y a plusieurs choses dans cet exemple de code. Ici, je déclare une classe *MaPremiereClasse*. Dans cette classe, je définis une fonction à l'aide du mot clé *def* qui s'appelle *__init__()*, c'est donc mon constructeur. Le mot *self* (soi) n'est pas un mot clé mais il est capital et il va bien falloir suivre son explication ! Il est obligatoirement présent en tant que premier argument du constructeur. Il permet de lier la fonction aux instances. Je l'expliquerai plus en détail dans la suite.

Le mot *self* (soi) fait référence à une instance de la classe. Nous n'en avons pas encore créé ici... En fait, lorsque vous écrivez une classe et que vous y définissez des choses, tout ce qui ce réfère à *self* appartiendra aux instances que vous créerez. C'est donc une référence générique à une instance. Vous comprendrez mieux à quoi cela sert lorsque nous définirons les attributs d'un objet.

Le mot *self* est l'équivalent du mot clé *this* d'autres langages de programmation orienté objet tels que C++ ou Java.

Maintenant que nous avons une classe définie, il est possible de créer des objets de type *MaPremiereClasse*. Pour cela, c'est très simple, il suffit de faire :

Code : Python

```
monObjet = MaPremiereClasse()
```

Si vous vérifiez le type de votre objet à l'aide de la fonction intégrée *type()*, vous pourrez constater que *monObjet* est bien du type *MaPremiereClasse*. Lorsque vous faites l'appel *MaPremiereClasse()*, c'est en fait la méthode *__init__()* qui est automatiquement appelée par l'interpréteur. Grâce à ce nom générique pour le constructeur, c'est donc le nom de votre classe qui vous permet de créer des objets du type que vous avez défini !

Vous remarquerez d'ailleurs qu'aucun argument n'a été donné au constructeur. En fait, la référence *self* est toujours donnée implicitement par l'interpréteur. Vous n'avez et ne devez en aucun cas donner une valeur pour *self*. Vous pouvez d'ailleurs essayer. L'interpréteur vous dira certainement que vous donnez deux arguments alors que *__init__()* n'en demande qu'un...

Si vous ne définissez aucun constructeur, il le sera automatiquement par défaut ! Il est donc tout à fait possible de déclarer une classe avec absolument rien dedans !

En résumé, les contraintes sur les constructeurs dont il faut absolument tenir compte pour que cela fonctionne sont les suivantes :

- **Le nom** : un constructeur s'appelle toujours *__init__()*
- **Les arguments** : il doit forcément y avoir *self* en premier argument
- **Le type de retour** : un constructeur ne doit rien retourner ! (En réalité, il retourne l'objet None qui est l'objet générique servant à représenté un objet qui n'existe pas.)

Pour le moment, cet objet ne contient rien du tout et n'est donc pas très utile. Afin d'en faire quelque chose qui puisse être réellement utilisé, il va falloir lui définir des attributs.

Définir les attributs d'instance

L'intérêt des objets est de modéliser des concepts ou des objets physiques caractérisés par un ensemble de données. Pour faire cela, un ensemble d'attributs d'instance peut être défini dans vos classes. Ils sont dit d'instance car ils auront une valeur différente (ou tout du moins un emplacement en mémoire différent) pour chacune de vos instances.

Code : Python

```
class Crayon :
    """
Cette classe permet de créer des Crayons qui pourront
servir à écrire ce tutoriel par exemple !
```

```
    """

        def __init__(self, couleur, quantiteEncre, tailleDuTrace) :
            """
    Constructeur de Crayon.
    couleur : La couleur d'écriture du crayon.
    quantiteEncre : La quantite d'encre dans le crayon.
    tailleDuTrace : La taille du tracé.
    """
            self.couleur = couleur
            self.quantiteEncre = quantiteEncre
            self.tailleDuTrace = tailleDuTrace
```

Dans cet exemple, je définis une classe *Crayon* et j'implémente son constructeur. Remarquez que vous pouvez donner autant de paramètres que vous le souhaitez à votre constructeur. Ici, j'en donne trois qui sont les données que je souhaite stocker dans les attributs d'instance de mon objet.

Le constructeur est en quelque sorte une fonction comme une autre. Il peut accepter les arguments avec une valeur par défaut mais aussi les listes et les dictionnaires d'arguments.

Vous remarquerez que le mot *self* a été utilisé à plusieurs reprises. Il permet de préciser qu'une donnée se trouve dans l'objet lui-même. Ainsi, ici, mes *Crayons* seront caractérisés par trois attributs : *couleur, quantiteEncre* et *tailleDuTrace*. Pour les associer à l'objet, je suis obligé de le préciser à l'aide de *self*. Ainsi, je différencie les variables appartenant à l'objet et les variables définies dans la fonction via les arguments. Les attributs d'instance sont donc des variables bien différentes de celles passées en paramètres du constructeur puisqu'elles sont reliées à *self* par la notation pointée !

Si je souhaite créer des Crayons et accéder à leur données, je peux donc faire ceci :

Code : Python

```
unCrayon = Crayon("Bleu", 20, 12)
unAutreCrayon = Crayon("Rouge", 20, 18)
encoreUnAutreCrayon = Crayon("Violet", unCrayon.quantiteEncre,
unAutreCrayon.tailleDuTrace)
```

Il est possible d'accéder aux données stockées dans un objet en utilisant la notation pointée. Chaque instance que je crée ici possède donc ses propres variables ! Vous pouvez d'ailleurs essayer de modifier la valeur d'un attribut d'une de vos instances et voir que cela ne modifie en rien les autres !

Dans la pratique, il n'est autorisé d'accéder directement aux attributs d'un objet uniquement dans la définition de la classe à l'aide du mot *self*. Ce que je fais là ne respecte pas le principe d'encapsulation et n'est donc là qu'à titre d'exemple !

Dans la pratique, le constructeur n'est pas la seule méthode acceptant la déclaration d'attributs d'instance mais si vous voulez vous éviter un bon paquet d'erreurs casse-têtes, je vous conseille de ne déclarer les attributs d'instance que dans le constructeur.

Méthodes d'instance

Maintenant que nous avons vu comment attribuer des données à un objet, nous allons voir comment y accéder et quoi faire de ces données. Nous allons créer ce que l'on appelle des méthodes d'instance. C'est-à-dire des méthodes qui s'appliquent directement à des instances de classe. Par exemple, la fonction *append()* est une méthode d'instance de la classe *list*. Afin de bien saisir l'intérêt du principe d'encapsulation, nous allons considérer une classe *Point* qui va nous permettre de créer des points dans un plan mais ici, nous les modéliserons en coordonnées polaires. Pour ceux qui ne seraient pas familiers avec les coordonnées polaires. Je vous invite à lire l'article wikipédia donné en lien. Pour les autres, un petit rappel ne fait pas de mal :

Citation : Wikipédia

Les coordonnées polaires sont, en mathématiques, un système de coordonnées à deux dimensions, dans lequel chaque point du plan est entièrement déterminé par un angle et une distance. Ce système est particulièrement utile dans les situations où la relation entre deux points est plus facile à exprimer en termes d'angle et de distance, voir par exemple le pendule.

Nous allons donc représenter les points dans notre modèle à l'aide d'un angle et d'une distance par rapport au centre.

L'utilisateur, lui, n'aura accès qu'aux coordonnées cartésiennes données par l'abscisse (largeur) et l'ordonnée (hauteur).

Code : Python

```
class Point :
    """
Classe permettant la création de points dans un plan.
"""

    def __init__(self, x, y) :
        """
Constructeur de Point.
x : Abscisse du point.
y : Ordonnée du point.
"""
        self.rho = math.sqrt(x**2 + y**2)
        if x != 0 :
            self.theta = math.atan(y/x)
        else :
            self.theta = math.pi / 2
```

Accesseurs et mutateurs

Afin de respecter le principe d'encapsulation, les attributs de nos objets ne doivent pas être accessibles directement. En Python, il n'existe aucun moyen d'empêcher l'utilisateur d'y accéder directement sauf en les nommant de façon très lourde voire incompréhensible et de façon non documentée. En général, cela dissuade assez vite mais nous allons espérer que les utilisateurs de nos classes souhaitent respecter le principe d'encapsulation. Pour rendre la manipulation des attributs tout de même possible, il existe ce que l'on appelle des **accesseurs** et des **mutateurs**.

Par convention, tous les accesseurs doivent avoir un nom commençant par **get** (obtenir) et tous les mutateurs par **set** (attribuer). Ainsi, dans l'exemple du point en coordonnées polaires, on peut rajouter des accesseurs et des mutateurs.

Code : Python

```
import math

class Point :
    """
Classe permettant la création de points dans un plan.
"""

    def __init__(self, x, y) :
        """
Constructeur de Point.
x : Abscisse du point.
y : Ordonnée du point.
"""
        self.rho = math.sqrt(x**2 + y**2)
        if x != 0 :
            self.theta = math.atan(y/x)
        else :
            self.theta = math.pi / 2

    def getX(self) :
        """
Accesseur de l'abscisse du point.
Retourne : L'abscisse du point.
"""
        return self.rho * math.cos(self.theta)

    def getY(self) :
        """
Accesseur de l'ordonnée du point.
Retourne : L'ordonnée du point.
```

```
        """
                return self.rho * math.sin(self.theta)

            def setX(self, x) :
                """
        Mutateur de l'abscisse du point.
        x : Le nouvel abscisse du point.
        """
                y = self.getY()
                self.rho = math.sqrt(x**2 + y**2)
                if x != 0 :
                    self.theta = math.atan(y / x)
                else :
                    self.theta = math.pi/2

            def setY(self, y) :
                """
        Mutateur de l'ordonnée du point.
        x : La nouvelle ordonnée du point.
        """
                x = self.getX()
                self.rho = math.sqrt(y**2 + x**2)
                if self.getX() != 0 :
                    self.theta = math.atan(y / x)
                else :
                    self.theta = math.pi/2
```

Dans cet exemple, mon modèle de point est en coordonnées polaires mais toute l'interface que je présente à l'utilisateur via les accesseurs et les mutateurs montre un point en coordonnées cartésiennes. Vous voyez donc ici, qu'il est tout à fait possible que le modèle réellement implémenté dans un objet ne corresponde pas du tout à ce à quoi vous accédez. J'ai volontairement pris cet exemple qui paraît compliqué alors que ce n'est pas nécessaire pour illustrer l'encapsulation.

Pour les non matheux, *sqrt()* (square root) retourne la racine carrée d'un nombre et *atan()* retourne un angle en radian à partir d'un nombre (ici le rapport entre le côté opposé et le côté contigu de l'angle sur lequel on travaille dans le triangle formé par le centre, le point et la projection sur l'axe horizontal du point...).

Maintenant, regardons de plus près ce que je fais. Toutes les fonctions que je définis ici prennent pour premier argument *self*. **Toutes les méthodes d'instance prennent *self* pour premier argument, c'est ainsi qu'elles sont définies**. Elles seront donc forcément associées à une instance de la classe. Pour y faire appel, il faudra alors utiliser la notation pointée.

Code : Python

```
point = Point(3,2)
print(point.getX())
print(point.getY())
point.setX(12)
point.setY(23.4)
print(point.getX())
print(point.getY())
```

En essayant ce petit morceau de code, vous pouvez constater que l'utilisateur n'a aucune raison de penser que le Point est en réalité représenté en coordonnées polaires ! Le principe d'encapsulation est donc parfaitement respecté ici. Grâce aux accesseurs et aux mutateurs, j'offre à l'utilisateur une interface claire pour utiliser ma classe *Point*. Il peut alors utiliser sans soucis ma classe et sans avoir à connaître ma modélisation.

Méthodes quelconques

Comme je l'ai dit précédemment, les accesseurs et les mutateurs sont des méthodes d'instance. Les autres méthodes d'instance se créent alors exactement de la même façon que les accesseurs et les mutateurs ! En déclarant une fonction dans une classe avec pour premier argument *self*, vous créez des méthodes d'instance applicables à vos objets. On peut par exemple rajouter des méthodes sur notre classe *Point*.

Code : Python

```
    def translater(self, x = 0, y = 0) :
        """
Translate le point de x en abscisse et de y
en ordonnée.
x = 0 : La projection sur l'axe horizontal de la
translation à effectuer.
y = 0 : La projection sur l'axe vertical de la
translation à effectuer.
"""
        self.setX(self.getX() + x)
        self.setY(self.getY() + y)

    def centrer(self) :
        """
Centre le point.
"""
        self.rho = 0
        self.theta = 0

    def calculerDistance(self, point) :
        """
Calcule la distance entre self et le
point donné en argument.
point : Un point dont on veut la distance par rapport
à self.
Retourne : La distance entre self et point.
"""
        return math.sqrt((self.getX() - point.getX())**2 + \
                         (self.getY() - point.getY())**2)
```

Vous voyez que je crée effectivement des fonctions de la même manière que pour les accesseurs et les mutateurs. Seule la convention de nommage des méthodes change. Je fais d'ailleurs appel aux accesseurs à l'intérieur même de ma classe mais je n'y suis pas contraint. Par exemple, dans la méthode *centrer()*, j'accède directement aux attributs de l'objet. Cela ne viole en aucun cas le principe d'encapsulation car je suis toujours dans la définition de la classe *Point*. Pour appeler ces méthodes, cela fonctionne toujours pas la notation pointée que vous connaissez.

Code : Python

```
point = Point(3,2)
point.translater(12,1)
print(point.getX())
print(point.getY())
point2 = Point(2.3, 8)
print(point.calculerDistance(point2))
print(point.calculerDistance(point))
```

Vous savez créer des classes de base. Nous allons voir par la suite que les classes peuvent aussi être utilisées pour stocker de l'information avec les attributs et les méthodes statiques ! Nous apprendrons aussi à faire en sorte que vos classes puissent être utilisées comme des types intégrés !

Créer des objets (2/2)

Vous savez créer des classes et les instancier mais ce n'est pas suffisant. Pour rendre vos programmes encore mieux structurés, vous allez apprendre à utiliser vos objets comme de vrais types intégrés !

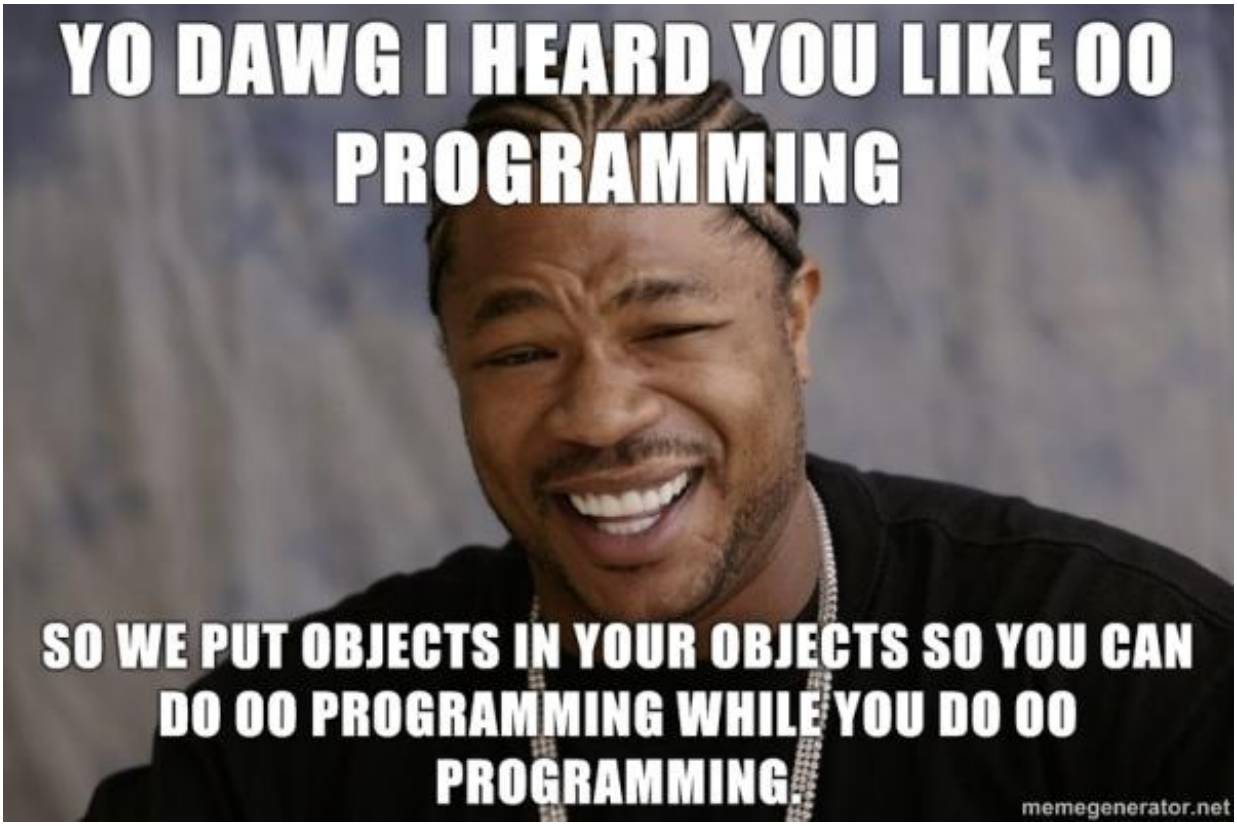
Pour cela, nous aborderons :

- La composition pour utiliser des objets dans vos objets
- Les attributs et les méthodes de classe pour se rapprocher de la programmation modulaire
- La surcharge des opérateurs pour simuler des types intégrés

Ca a l'air sympathique hein ? Bien entendu, ce que vous apprendrez ici n'est en rien obligatoire pour faire de la POO mais cela vous sera certainement très utile pour la suite alors soyez attentifs ! Les deux du fond de la classe, c'est la première et dernière fois que je vous rappelle à l'ordre !

Composition

La composition consiste en l'utilisation d'objets en tant qu'attribut d'objets... En clair, vous pouvez utiliser vos objets comme des types de variable habituelle ! Dans la culture web, cela pourrait s'exprimer ainsi :



Concrètement, à quoi cela peut-il servir ? Et bien vous pouvez créer un modèle entièrement objet sans aucun problème. Prenons l'exemple d'un livre. C'est un objet. Il est lui-même composé de différents objets, la couverture et les pages. Avec la composition, vous pouvez créer des relations entre objets du type "appartient à". Ainsi, les pages appartiennent à un livre.

Dans la pratique, on pourrait avoir quelque chose de cette forme :

Code : Python - Classe Page

```
class Page :
    """
Objet page contenant un texte.
"""

    def __init__(self, texte, numero) :
        """
Constructeur de page.
texte : Le texte de la page.
numero : Le numéro de page.
"""
```

```
        self.texte = texte
        self.numero = numero

    def lirePage(self) :
        """
Affiche le contenu de la page.
"""
        print(self.texte)

    def getNumero(self) :
        """
Accesseur du numéro de page.
Retourne : Le numéro de page.
"""
        return self.numero

    def getTexte(self) :
        """
Accesseur du texte de la page.
Retourne : Le texte de la page.
"""
        return self.texte
```

Code : Python - Classe Livre

```
class Livre :
    """
Modélisation d'un livre.
"""

    def __init__(self, texte, titre, auteur) :
        """
Constructeur du livre.
texte : Le contenu du livre.
titre : Le titre du livre.
auteur : L'auteur du livre.
"""
        self.titre = titre
        self.auteur = auteur
        #Création des pages (10 lignes par page) :
        self.pages = []
        nbLignes = 0
        page = ""
        numero = 1
        for ligne in texte.split("\n") :
            page += ligne + "\n"
            nbLignes += 1
            if nbLignes == 10 :
                self.pages.append(Page(page, numero))
                nbLignes = 0
                page = ""
                numero += 1

    def getPage(self, numero) :
        """
Accesseur des pages.
numero : Le numéro de page, commence ≠ 1.
"""
        return self.pages[numero - 1]

    def lirePage(self, numero) :
        """
Lire une page en particulier.
numero : Le numéro de page, commence ≠ 1.
"""
        self.pages[numero - 1].lirePage()
```

Dans cet exemple de livre, on utilise une liste de Pages comme attribut des Livres. Vous avez un excellent exemple de composition avec des Pages contenues dans un objet *list* lui-même contenu dans un objet Livre. Toutes les classes que vous créez peuvent être utilisées au même titre que les types de données intégrés au langage Python.

Attributs et méthodes de classe

Les classes peuvent être utilisées comme des sortes de modules. Nous allons voir qu'il est possible d'y définir des variables et des fonctions qui pourront être utilisés comme ce que vous aviez fait jusqu'à maintenant avec des modules.

Attributs de classe

Tout d'abord, il est possible de définir des attributs de classe qui ne seront pas associés à une instance en particulier mais à toute la classe. Toutes les instances partageront donc cet attribut. Pour créer de tels attributs, il suffit de les déclarer directement dans une classe.

Code : Python

```
class Test :
    """Doc"""

    variableTest = 0
```

Pour les codeurs Java ou C++, ces attributs sont tout simplement des variables statiques.

En faisant cela, on crée une variable globale au sein de la classe *Test*. Il est possible d'accéder à ce type d'attributs soit via la classe soit via une instance. Mais attention, d'une instance sur l'autre, la variable sera exactement la même ! Faisons un petit essai en shell.

Code : Python

```
>>> class Test :
 variable = 0


>>> t = Test()
>>> Test.variable
0
>>> t.variable
0
>>> Test.variable += 1
>>> t.variable
1
>>> Test.variable
1
```

Je déclare ici ma classe de test. J'instancie cette classe puis accède à la variable de classe via la classe puis via une instance. Vous pouvez remarquer qu'ensuite je modifie la valeur de cette variable en y accédant via la classe mais la valeur est bel et bien modifiée aussi pour l'instance ! C'est donc bien un attribut de classe !

En permettant l'accès des attributs de classe via une instance, Python crée une ambiguïté sur les noms des attributs de classe et d'instance. L'interpréteur ne pourra différencier ces deux types d'attributs ! Essayez donc de toujours donner des noms différents aux attributs de classe et aux attributs d'instance ! Cela vous évitera quelques crises de nerfs...

L'intérêt de ces attributs réside principalement dans la définition de constantes qui peuvent être utilisées dans la classe ou à l'extérieur. Par exemple, dans le jeu de la vie, j'ai défini une liste des formes possibles. Un attribut de classe est un excellent moyen de proposer une liste de constantes correspondant à une liste de choix possibles. Mais ces attributs peuvent aussi être utilisés pour donner la version du programme, compter le nombre d'instances créées et encore bien d'autres choses !

Méthodes de classe

Comme il est possible de définir des attributs de classe, il est possible de définir des méthodes de classe. Ces fonctions sont exactement l'équivalent sous forme de fonction des attributs de classe. Elles pourront être appelées soit via une instance soit via la classe. Une telle fonction prend comme premier argument non pas une instance mais une classe. En général, cet argument sera nommé *cls*. Pour en déclarer il faut faire :

Code : Python

```
class Test :
    """Doc"""

    def fonction(cls) :
        """Doc"""
        print("Je suis une méthode de classe !")
        print("J'appartiens à la classe %s" % cls.__name__)

    fonction = classmethod(fonction)

t = Test()
t.fonction()
Test.fonction()
```

Dans cet exemple, je déclare une fonction que je dois ensuite déclarer comme étant un attribut de classe. Mais c'est un attribut de classe particulier qui peut être appelé. Pour rendre cela possible, j'utilise la fonction intégrée *classmethod()* qui s'applique à une fonction et qui permet de préciser que l'attribut sera bel et bien une fonction de classe. De cette manière, je peux ensuite appeler la fonction via la classe ou via une instance.

Vous remarquerez d'ailleurs que je fais appel à un attribut spécial de ma classe : *__name__*. Toutes les classes possèdent un ensemble d'attributs spécifiques contenant certaines informations à leur sujet. Vous trouverez tout ce qu'il y a à savoir là-dessus dans la documentation.

Comme pour les attributs de classe, l'interpréteur ne fera pas la différence entre une méthode d'instance et une méthode de classe... Ne donnez donc pas le même nom à vos fonctions même si elles ne sont pas du même type ! Cela vous évitera de belles erreurs...

Méthodes statiques

Les méthodes statiques seront appelées de la même manière que les méthodes de classe mais ne prennent aucun argument obligatoire. Leur déclaration est relativement similaire aux méthodes de classe puisque la seule différence réside dans le fait d'appeler la fonction intégrée *staticmethod()* au lieu de *classmethod()*.

Code : Python

```
class Test :
    """Doc"""

    def fonction() :
        """Doc"""
        print("Je suis une fonction statique !")

    fonction = staticmethod(fonction)

Test.fonction()
t = Test()
t.fonction()
```

Avec cet exemple, vous pouvez constater que les deux types d'appels fonctionnent. La seule différence est donc bien dans le fait que les méthodes statiques ne prennent pas pour premier argument une classe.

Avec les attributs et les méthodes de classe, vous pouvez faire de vos classes de véritables modules ! Cela vous permet ainsi de créer des ensembles cohérents de données et de fonctions tous organisés autour d'un même thème représenté par la classe. Pour ceux qui ont l'habitude de ne programmer qu'en objet, vous ne serez donc pas déboussolés.

Surcharge des opérateurs

Dans la plupart des langages objets, il est possible de surcharger les opérateurs. Mais d'abord, qu'est-ce que cela veut dire ? La surcharge des fonctions (*override* en anglais) consiste en la réécriture de celle-ci pour modifier leur comportement. Dans notre cas, ce qui va nous intéresser, c'est la surcharge des opérateurs intégrés tels que l'addition, la soustraction, la conversion en chaîne de caractères etc...

Mais euh, comment je peux réécrire ces fonctions si elles ne le sont pas de base ?! Quand je crée un objet, je ne peux pas en additionner plusieurs ! 

Et c'est là toute la magie de Python. Tous les objets que vous créez peuvent en fait surcharger des méthodes qui ne sont pas implémentées explicitement. D'ailleurs vous le faites déjà lorsque vous écrivez la méthode *__init__()*. En réalité, vous la surchargez pour qu'elle ne fasse pas rien.

Maintenant que l'on sait que l'on peut surcharger les opérateurs, comment ça marche ? Et bien, si vous regardez la documentation à ce sujet, vous pouvez remarquer qu'il existe pas mal de choses. En effet, il est possible d'émuler des conteneurs tels que les listes mais aussi des types numériques. Toutes les fonctions à surcharger sont entourées d'un double espace souligné (underscore, '_') et portent toutes un nom plus ou moins explicite du moment que l'on parle anglais.

Pour l'exemple, nous allons créer une classe pour représenter les temps et nous essayerons d'effectuer les opérations de base sur celle-ci. Commençons par déclarer une classe Temps.

Code : Python

```
class Temps :
    """
Classe permettant de créer des temps.
"""

    def __init__(self, temps) :
        """
Constructeur de temps !
temps : Le temps en secondes.
"""
        self.temps = temps
```

Maintenant que l'on peut créer des temps, nous allons essayer de les afficher correctement en surchargeant la méthode les convertissant en chaîne de caractères. Cette méthode, c'est *__str__() (string)* et c'est elle qui est automatiquement appelée lorsque vous utilisez une variable dans *print()* ou *input()* pour l'afficher dans la console. Elle doit donc retourner une chaîne de caractères.

Code : Python

```
def __str__(self) :
    """ Surcharge de la conversion en chaîne de caractères."""
    jour = self.temps//86400
    heures = (self.temps%86400)//3600
    minutes = (self.temps%3600)//60
    secondes = (self.temps%60)
    texte = ""
    if jour :
        texte += "%ij" % jour
    if heures :
        texte += "%ih" % heures
    if minutes :
        texte += "%imin" % minutes
    texte += "%is" % secondes
    return texte
```

Avec cette fonction, dès que vos temps devront être convertis en chaîne de caractères, ils le seront automatiquement avec un appel de la fonction *__str__()*. Le petit test tout simple consiste à créer plusieurs temps et à essayer de les afficher dans la console.

Code : Python

```
liste = [Temps(12), Temps(342), Temps(8456), Temps(2348509)]
for temps in liste :
    print(temps)
```

Vous devriez constater que vos temps s'affichent de la manière dont nous l'avons défini dans la fonction *__str__()*. Si c'est le cas, c'est que tout a bien fonctionné ! Voyons maintenant si l'on peut comparer des temps. Nous allons surcharger les méthodes de comparaison. Ici, ce ne sera pas très compliqué. Les méthodes à surcharger sont *__lt__()* (*Lower Than*, plus petit que), *__le__()* (*Lower or Equal*, plus petit ou égal), etc...

Code : Python

```
def __lt__(self, autre) :
    """Surcharge de <"""
    return self.temps < autre.temps

def __le__(self, autre) :
    """Surcharge de <="""
    return self.temps <= autre.temps

def __eq__(self, autre) :
    """Surcharge de =="""
    return self.temps == autre.temps

def __ne__(self, autre) :
    """Surcharge de !="""
    return self.temps != autre.temps

def __gt__(self, autre) :
    """Surcharge de >"""
    return self.temps > autre.temps

def __ge__(self, autre) :
    """Surcharge de >="""
    return self.temps >= autre.temps
```

Avec le modèle choisi pour représenté les temps, la comparaison est assez triviale puisqu'il suffit de comparer les deux temps proprement dits. Sachez que vous auriez pu créer une classe Temps où vous auriez stocké les jours, les heures, les minutes et les secondes séparément. La comparaison n'aurait donc pas été directe. Ceci étant fait, vous pouvez maintenant utiliser les opérateurs de comparaison sur vos temps comme s'ils étaient des types intégrés au langage Python. Cela vous permettra de simplifier vos codes en écrivant des opérations simples sur vos objets plutôt qu'en faisant des appels de fonctions habituels. Vous pouvez faire quelques petits tests.

Code : Python

```
liste = [Temps(12), Temps(342), Temps(8456), Temps(2348509)]
print("12s < 342s : %s" % (liste[0] < liste[1]))
print("12s > 342s : %s" % (liste[0] > liste[1]))
```

Avec ces exemples, vous avez vu qu'il est possible de surcharger les opérateurs de comparaison et de conversion de type avec *__str__()* par exemple. Il est aussi possible de surcharger les opérations de base comme l'addition ou la soustraction.

Code : Python

```
def __add__(self, autre) :
    """Surcharge de +"""
    return Temps(self.temps + autre.temps)

def __sub__(self, autre) :
    """Surcharge de -"""
```

```
        return Temps(self.temps - autre.temps)
```

Ainsi, vous pouvez maintenant additionner ou soustraire des temps sans aucun souci ! Vous pouvez d'ailleurs utiliser la syntaxe simplifiée '+=' ou '-=' comme avec les types intégrés de Python si vous avez surchargé les opérations concernées.

Code : Python

```
liste = [Temps(12), Temps(342), Temps(8456), Temps(2348509)]
temps = liste[0] + liste[1]
print(temps)
temps += Temps(5668)
print(temps)
```

J'espère que vous comprenez l'intérêt de la surcharge des opérateurs. Cela vous permettra de simplifier votre code et de le rendre plus lisible lorsque vous devez utiliser couramment des opérations simples. Cependant, vous êtes laissé entièrement libre sur l'implémentation des opérateurs. Vous pouvez créer de véritables fonctions mais essayez de faire en sorte qu'elles soient instinctives et correspondent au symbole que vous utilisez si vous ne voulez pas perdre l'utilisateur de vos classes voire vous-même !

Si vous voulez émuler des conteneurs du type des listes ou des dictionnaires, vous devriez lire la partie appelée "Emulating containers types" de la documentation. Vous y trouverez de quoi surcharger les méthodes d'accès aux éléments par indice ou clé par exemple.
Vous savez maintenant utiliser créer des classes et les utiliser de façon basique. Avant de passer à la suite dans la POO, je vous propose un petit TP facile pour vous exercer !

Partie en chantier !

La programmation orientée objet n'a maintenant plus de secrets pour vous ! Mais avec une interface graphique, c'est pas mieux ? Si, et c'est pour ça que la prochaine partie va vous les introduire (je vous y prends bande de petits cochons ) !

Partie 4 : Annexes

Des bonnes pratiques

Cette petite partie est consacrée aux bonnes habitudes à prendre lorsque l'on programme pour rendre son code compréhensible par d'autres programmeurs mais aussi par soi-même lorsque l'on revient après plusieurs mois sur un code que l'on a fait. Ce sera rapide, mais quelques petits rappels ne font jamais de mal !

Nommer

Des noms explicites...

La première chose très importante pour rendre son code lisible est d'utiliser des noms explicites pour vos variables (non pas explicite dans ce sens là...). J'entends par là, que tout les noms que vous utilisez doivent être parlants ! Par exemple, utiliser des noms de hamburger ne vous aidera pas vraiment à comprendre le fonctionnement du code lorsque vous le relirez... Bien sûr, ça peut être très bien si vous modéliser un fast food...

Vous n'êtes pas limité dans le nombre de caractères pour les noms de variables ou de fonctions...

Prenons un petit exemple simple, essayez de deviner ce que fais le code ci-dessous. Il est écrit une fois avec des noms débiles, et une fois avec des vrais noms de variables :

Code : Python

```
def peuh(var) :
 cheeseburger = 0
 if var != 0 :
  cheeseburger = int(math.log(var,10))
 if cheeseburger < 0 :
  cheeseburger -= 1
 return cheeseburger

print(peuh(0.0003843))
```

Secret (cliquez pour afficher)

Code : Python

```
def getPower(number) :
 """
Return the 10 power of the number given in argument.
number : the number to test.
"""
 result = 0
 if number != 0 :
  result = int(math.log(number,10))
 if result < 0 :
  result -= 1
 return result

print(getPower(0.0003843))
```

De la normalisation...

Un autre excellent moyen de rendre son code compréhensible est de garder toujours les mêmes normes de nommage des variables et des fonctions. Ainsi, j'utilise la norme suivante :

- Tous les noms de variables et de fonctions commencent par un lettre minuscule
- Les noms de variables et de fonctions ne sont composés que de lettres et de chifffres
- Chaque mot au sein d'un nom prend une majuscule (sauf le premier)
- Les noms des constantes sont écrits uniquement en majuscules

La norme à utiliser est libre mais doit être constante pour rendre le code plus lisible et facilement compréhensible.

Documenter

C'est peut-être le point le plus important de tous ! Si votre code est bien documenté, alors il n'est pas nécessaire de lire autre chose que la documentation pour l'utiliser ! La devise d'une très grande entreprise d'informatique est :

Citation

> *Si vous n'êtes pas capable de comprendre un code en ne lisant que sa documentation, c'est qu'il est bon à jeter...*

Donc faites plaisir aux gens qui vous reliront et à vous-même en documentant du mieux possible vos codes. Pour générer la documentation au format html par exemple, il existe différents programmes:

- **Pydoc** : programme officiel générateur de documentation
- **Doxygen** : générateur de documentation pour tous les codes prenant en compte différents formats de documentation (j'ai un faible pour celui-ci)
- Et bien d'autres encore...

Ce genre de documentation vous permet de comprendre rapidement le fonctionnement d'objets ou de modules. Doxygen génère notamment les arbres d'héritage si nécessaire, ce qui est relativement pratique pour avoir une vue d'ensemble d'une bibliothèque...

Références

Vous pensez que je ne me suis aidé de rien du tout pour vous écrire un tutoriel complet qui ressemble à quelque chose ? Je ne pense pas cela aurait été possible sans d'excellentes références avec lesquelles j'ai moi-même appris à programmer ! Ici, vous trouverez tous les ouvrages ou presque j'ai pu utiliser (j'espère en faire une liste la plus exhaustive possible...) mais aussi les ressources web à connaître qui ne pourront que vous aider dans votre long périple au coeur de Python.

Livres

Parce que nous avons souvent tendance à les délaisser alors qu'ils sont le plus souvent une référence incontournable, les livres sont capitaux lorsque l'on veut apprendre quelque chose. C'est en allant chez le libraire du coin que j'ai pu trouver mes premiers livres de programmation et j'espère que vous en ferez de même. Pour ma part, j'ai utilisé :

- Python : Le guide de survie, Brad Dayley (CampusPress, 16€)
- Au coeur de Python (vol 1 et 2), Whesley J. Chun (CampusPress, 48€ et 35€)

Vous pouvez vous procurez ses bouquins sur internet ou chez n'importe quel libraire !

Ressources web

Programmer sans internet c'est un peu comme vouloir taper au clavier sans main, c'est difficile... C'est donc aussi grâce à des ressources web que j'ai pu apprendre à programmer et écrire ce tutoriel. Les ressources que j'ai utilisées sont donc :

- Apprendre à programmer en Python, Gérard Swinnen : cours complet trouvable en format pdf sur le web
- Developpez.net : cours et forum de discussion pour les programmeurs
- Commentcamarche.net : parce qu'il est rare d'avoir un problème auquel personne n'ait répondu

Et il y en a bien d'autres que j'ai pu utiliser ponctuellement pour des idées d'exercices ou pour trouver des réponses à des problèmes spécifiques.

Quelques idées d'application

Savoir programmer, c'est bien beau mais il faut bien trouver quelque chose à programmer ! Pour ça, je vous suggère quelques lectures intéressantes que j'ai pu avoir voire que j'ai encore (et oui... Je n'ai pas forcément le temps de ne faire que de l'informatique... ). Donc, voici quelques références sympathiques qui peuvent vous donner des idées :

- Intelligence artificielle, S. Russel et P. Norvig (Pearson Education, 72€)
- Techniques de hacking, Jon Erickson (Pearson, 32€)

J'espère que cela vous inspirera et que cela vous occupera pour un bout de temps !

Ce tutoriel n'est pas encore terminé, loin de là ! Sa construction ne fait que commencer et je vais profiter d'un peu de temps libre durant mes vacances pour essayer de l'avancer au plus vite... Alors soyez patients, entraînez-vous au maximum et revenez jeter un coup d'oeil de temps en temps !

Dans la partie programmation orientée objet vous verrez :
- Ce qu'est un objet
- Que vous savez déjà les utiliser
- Comment les créer vous-même
- L'héritage et le polymorphisme
- Le tout accompagné d'exemples et de TP !

Dans la partie sur l'introduction aux GUI, vous verrez :
- Ce qu'est une GUI
- Comment en créer avec Tkinter
- La programmation évènementielle
- Comment créer ses propres composants graphiques
- Une petite intro à PyGTK et à PyQt
- Et bien sûr toujours des exercices et des TP

Dans la partie sur les modules de base, vous verrez :
- Quelques fonctionnalités avancées du langage Python
- La gestion du temps dans vos applications
- La gestion des bases de données en Python
- La gestion du multithreading

- Etc...

Dans la partie pour aller plus loin, il y aura des introductions à :
- PyGame
- SciPy et PyLab
- Panda3D
- Blender
- Etc...

Le but de ce cours est donc d'être le plus complet possible en vous offrant la possibilité d'en apprendre un peu plus sur les possibilités de ce magnifique langage qu'est Python ! Bien entendu, tout ce cours est basé en grande partie sur les expériences que j'ai pu avoir avec ce langage et d'autres. J'espère qu'il vous sera utile !

J'attends impatiemment vos commentaires pour l'améliorer !