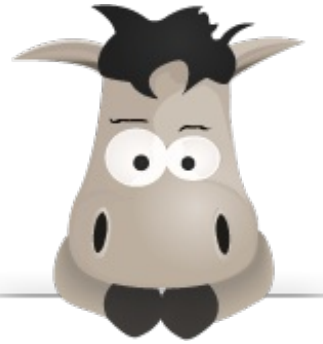

Apprendre Python !

Par prolixe

www.siteduzero.com

Dernière mise à jour le 8/03/2012

Sommaire

Apprendre Python !

Par prolixe
Mise à jour : 16/04/2011
Difficulté : Facile

Ce tutoriel a pour but de vous initier au langage de programmation Python. Et comme le veut la coutume ici-bas, on démarre de zéro, dans la joie et la bonne humeur !

La syntaxe claire et relativement intuitive de ce langage en fait un candidat idéal dans le cadre d'une introduction à la programmation. Ainsi, si vous n'avez jamais programmé en quelque langage que ce soit, si vous ne savez que très vaguement ce que cela signifie, Python est, me semble-t-il, un bon choix pour commencer votre apprentissage. Bonne lecture !

Avantages de Python :

- facile à apprendre, à lire, à comprendre et à écrire ;
- portable (fonctionne sous de nombreux systèmes d'exploitation) ;
- adapté aussi bien pour des scripts, des petits ou gros projets ;
- doté d'une façade objet bien conçue et puissante ;
- possède une communauté active autour du langage ;
- et j'en passe…

Un grand merci à 6pri1 pour sa relecture attentive et sa patience. Un merci tout aussi cordial à Nathan21 et Sergeswi qui ont fourni les icônes du tutoriel.

Ce cours vous plaît ?

Si vous avez aimé ce cours, vous pouvez retrouver le livre "*Apprenez à programmer en Python*" du même auteur, en vente sur le Site du Zéro, en librairie et dans les boutiques en ligne. Vous y trouverez ce cours adapté au format papier avec une série de chapitres inédits.

Plus d'informations

Partie 1 : Introduction à Python

Cette partie consiste en une introduction à Python et ses principaux mécanismes. Vous y apprendrez :

- Ce qu'est exactement Python ;
- Comment installer Python ;
- Comprendre la syntaxe et les mécanismes de base de ce langage.

Ne vous alarmez pas outre mesure si vous êtes déjà perdus dans le titre des sous-parties. J'ai promis que je commencerai de zéro, et je tiendrai cette promesse, autant que faire se peut. Commencez donc par le commencement, et continuez dans cette voie, c'est garanti sans douleur… du moins sans douleur excessive .

Qu'est-ce que Python ?

Bonjour et bienvenue !

Alors, vous avez décidé d'apprendre Python, et je ne peux que vous en féliciter. J'essayerai d'anticiper sur vos questions et de ne laisser personne en arrière. En fonction de votre niveau en informatique et en programmation, vous risquez d'ingérer une plus ou moins grande quantité d'informations. Cependant, et sauf si vous avez déjà des connaissances avancées dans ce langage, je vous conseille de suivre attentivement cette partie sans omettre un seul chapitre. L'heure n'est pas encore à la sélection.

Et d'abord, c'est quoi Python ?

Je ne m'avance pas trop, c'est une question assez facile à deviner si vous n'avez jamais entendu parler de Python auparavant.

Python est un langage de programmation, et pour bien commencer, je vais d'abord vous expliquer ce qu'est un langage de programmation. Je vais ensuite vous expliquer brièvement l'histoire de Python, pour que vous sachiez au moins d'où ça vient ! Rassurez-vous, il n'y aura pas de QCM pour ce premier chapitre, mais encore une fois je vous conseille de ne pas passer au suivant sans l'avoir lu.

On attaque tout de suite !

Un langage de programmation ? Qu'est-ce que c'est ?

La communication humaine

Non, ceci n'est ni une explication biologique ni philosophique, ne partez pas !
Très simplement, si vous arrivez à comprendre ces suites de symboles étranges et déconcertants que sont les lettres de l'alphabet, c'est parce que nous respectons une certaine convention, dans le langage, et dans l'écriture. En français, il y a des règles de grammaire et d'orthographe, je ne vous apprends rien. Vous communiquez en connaissant, plus ou moins consciemment, ces règles et en les appliquant plus ou moins bien, selon les cas.
Cependant, ces règles peuvent être aisément contournées : personne ne peut prétendre connaître l'ensemble des règles de la grammaire et de l'orthographe française, et peu de gens s'en soucient. Après tout, même si vous faites des fautes, les personnes avec qui vous communiquez pourront facilement vous comprendre.
Quand on communique avec un ordinateur cependant, c'est très différent.

Mon ordinateur communique aussi !

Eh oui, votre ordinateur communique sans cesse avec vous, et vous communiquez sans cesse avec lui. D'accord, il vous dit très rarement qu'il a faim, que l'été s'annonce caniculaire et que le dernier disque de ce groupe très connu était à pleurer.
Il n'y a rien de magique si, quand vous cliquez sur la petite croix en haut à droite de l'application en cours, elle comprend qu'elle doit se fermer.

Le langage machine

En fait, votre ordinateur se fonde aussi sur un langage pour communiquer avec vous ou avec lui-même. Les opérations qu'un ordinateur peut effectuer à la base sont des plus classiques et consistent en l'addition de deux nombres, leur soustraction, leur multiplication, leur division, entière ou non. Et pourtant, ces cinq opérations suffisent amplement à faire tourner les logiciels de simulation les plus complexes, ou les jeux super-réalistes.

Tous ces logiciels fonctionnent en gros de la même façon :

- une suite d'instructions écrites en langage machine compose le programme ;
- lors de l'exécution du programme, ces instructions décrivent à l'ordinateur ce qu'il faut faire (l'ordinateur ne peut pas le deviner).

Une liste d'instructions ? Qu'est-ce que c'est encore, ça ?

En schématisant volontairement, une instruction pourrait demander au programme de se fermer si vous cliquez sur la croix en haut à droite de votre écran, ou de rester en tâche de fond si tel est son bon plaisir. Toutefois, en langage machine, une telle action demande à elle seule un nombre assez important d'instructions.
Mais bon, vous pouvez vous en douter, parler avec l'ordinateur en langage machine, qui ne comprend que le binaire, ce n'est ni très enrichissant, ni très pratique, et en tous cas pas très marrant .
On a donc inventé des langages de programmation pour faciliter la communication avec l'ordinateur.

Le langage binaire est uniquement constitué de 0 et de 1. « 01000010011011110110111001101010011011110111010101110010 » par exemple signifie « Bonjour ». Bref, autant vous dire que discuter avec un ordinateur en binaire peut être long (surtout pour vous).

Les langages de programmation

Les langages de programmation sont des langages bien plus faciles à comprendre pour nous, pauvres êtres humains que nous sommes. Le mécanisme reste le même, mais le langage est bien plus compréhensible. Au lieu d'écrire les instructions dans une suite assez peu intelligible de 0 et de 1, les ordres donnés à l'ordinateur sont écrits dans un « langage », souvent en anglais, avec une syntaxe particulière qu'il est nécessaire de respecter. Mais avant que l'ordinateur puisse comprendre ce langage, celui-ci sera traduit pour lui en langage machine.
En gros, le programmeur « n'a qu'à » écrire des **lignes de code** écrites dans le langage qu'il a choisi, les étapes suivantes sont automatisées pour permettre à l'ordinateur de les décoder.

Il existe un grand nombre de langages de programmation, et Python en fait partie. Il n'est pas nécessaire pour le moment de donner plus d'explications sur ces mécanismes très schématisés. Si vous n'avez pas réussi à comprendre les mots de vocabulaire et l'ensemble de ces explications, cela ne vous pénalisera pas pour la suite. Mais je trouvais intéressant de donner ces précisions quant aux façons de communiquer avec son ordinateur.

Pour la petite histoire

Python est un langage de programmation, dont la première version est sortie en 1991. Créé par **Guido van Rossum**, il a voyagé du Macintosh de son créateur travaillant à cette époque au *Centrum voor Wiskunde en Informatica*, aux Pays-Bas, jusqu'à se voir attribuer une organisation à but non lucratif particulièrement dévouée, la **Python Software Foundation**, créée en 2001. Ce langage porte ce nom directement dérivé de celui de la série télévisée Monty Python .

Un langage de programmation interprété

Eh oui, vous allez devoir patienter encore un peu, car j'ai deux ou trois choses à vous expliquer encore, et je suis persuadé qu'il est important de savoir un minimum ces détails qui peuvent sembler peu pratiques de prime abord.
Python est un langage de programmation **interprété**, c'est-à-dire que les instructions que vous lui enverrez seront « transcrites » en langage machine au fur et à mesure de leur lecture. D'autres langages (comme le C / C++) sont dits « langages **compilés** » car, avant de pouvoir les exécuter, un logiciel spécialisé se charge de transformer le code du programme en langage machine. On appelle cette étape la « **compilation** ». À chaque modification du code, il faudra rappeler une étape de compilation.

Les avantages d'un langage interprété sont sa simplicité (on ne passe pas par une étape de compilation avant d'exécuter son programme) et la portabilité (un langage tel que Python est censé marcher aussi bien sous Windows, Linux et Mac OS, et on ne devrait avoir à effectuer aucun changement dans le code pour le passer d'un système à l'autre). Cela ne veut pas dire que les langages compilés ne sont pas portables, loin de là ! Mais des compilateurs différents doivent être utilisés et certaines instructions ne sont pas compatibles avec tous les systèmes, voire se comportent différemment.

En contre-partie, un langage compilé se révélera bien plus rapide qu'un langage interprété (la traduction à la volée de votre programme ralentit l'exécution), bien que cette différence tende à se faire de moins en moins sentir au fil des améliorations. De plus, Python devra être installé sur le système d'exploitation que vous utilisez pour que l'ordinateur puisse comprendre votre code.

Différentes versions de Python

Lors de la création de la Python Software Foundation, en 2001, et durant les années qui ont suivi, le langage Python est passé par une suite de versions que l'on a englobées dans l'appellation Python 2.x (2.3, 2.5, 2.6…). Depuis le 13 février 2009, la version 3.0.1 est disponible. Cette version casse la **compatibilité ascendante** qui prévalait lors des dernières versions.

Compatibilité quoi ?

Quand un langage de programmation est mis à jour, les développeurs se gardent bien de supprimer ou de trop modifier d'anciennes fonctionnalités. L'intérêt est qu'un programme qui marche sous une certaine version marchera toujours avec la nouvelle version en date. Cependant, la Python Software Foundation, observant un bon nombre de fonctionnalités obsolètes, implémentées plusieurs fois… a décidé de nettoyer tout le projet. Un programme qui tourne à la perfection sous Python 2.x devra donc être mis à jour un minimum pour marcher de nouveau sous Python 3. C'est pourquoi je vais vous conseiller ultérieurement de télécharger et d'installer la dernière version en date de Python. Je m'attarderai en effet sur les fonctionnalités de Python 3, et certaines d'entre elles ne seront pas accessibles (ou pas sous le même nom) dans les anciennes versions.

Ceci étant posé, tous à l'installation !

Installer Python

L'installation de Python est un jeu d'enfant, aussi bien sous Windows que sous les systèmes Unix.

Sous Windows

Rendez-vous sur le site de Python :

1. Cliquez sur le lien `Download` dans le menu principal de la page.
2. Sélectionnez la version de Python que vous souhaitez utiliser (je vous conseille la dernière en date).
3. On vous propose un (ou plusieurs) lien(s) vers une version Windows, sélectionnez celle qui conviendra à votre processeur. Si vous avez un doute, téléchargez une version « x86 ».
   Si votre ordinateur vous signale qu'il ne peut exécuter le programme, essayez une autre version de Python.
4. Enregistrez puis exécutez le fichier d'installation, et suivez les étapes, ce n'est ni très long ni très difficile.
5. Une fois l'installation complétée, vous pouvez vous rendre dans le menu `Démarrer > Tous les programmes`. Python devrait être visible dans cette liste. Nous verrons bientôt comment le lancer, pas d'impatience…

Sous Linux

Python est pré-installé sur la plupart des distributions Linux. Cependant, il est possible que vous n'ayez pas la dernière version en date. Pour le vérifier, tapez dans un terminal la commande `python -V`. Cette commande vous retourne la version de Python actuellement installée sur votre système. Il est très probable que cette version soit *2.X*, comme 2.6 ou 2.7, pour des raisons de compatibilité. Dans tous les cas, je vous conseille d'installer Python 3.X, la syntaxe est très proche de Python 2.X mais diffère quand même… 

Rendez-vous sur le site de Python, cliquez sur `download`, et téléchargez la version de Python (actuellement « *Python 3.2 compressed source tarball (for Linux, Unix or OS X)* »). Ouvrez un terminal, puis rendez-vous dans le dossier où se trouve l'archive :

1. Décompressez l'archive en tapant : `tar -jxvf Python-3.2.tar.bz2` (cette commande est bien entendu à changer en fonction de la version et du type de compression).
2. Attendez quelques instants que la décompression soit complétée, puis rendez-vous dans le dossier créé dans le répertoire courant (`Python-3.2` dans mon cas).
3. Exécutez le script `configure` en tapant `./configure` dans la console.
4. Une fois que la configuration s'est déroulée, il n'y a plus qu'à compiler en tapant `make altinstall`. Cette commande compilera Python et va créer automatiquement les liens vers la version installée. Grâce à `altinstall`, vous pouvez être sûrs que la version que vous installez ne rentrera pas en conflit avec une autre déjà installée sur votre système.

Sous Mac OS

Téléchargez la dernière version de Python sur le site officiel. Ouvrez votre fichier `.dmg` et double-cliquez sur le paquet d'installation `Python.mpkg`

Un assistant d'installation s'ouvre, laissez-vous guider : Python est maintenant installé !

Merci à Etienne-02 pour ses explications.

Lancer Python

Ouf ! Voilà qui est fait !

Bon, en théorie, on commence à utiliser Python dès le prochain chapitre, mais pour que vous soyez un peu récompensés de votre installation exemplaire, voici les différents moyens d'accéder à la ligne de commande Python que nous allons tout particulièrement étudier dans les prochains chapitres.

Sous Windows

Vous avez plusieurs façons d'accéder à la ligne de commande Python, la plus évidente étant de se rendre dans le menu `Démarrer > Tous les programmes > Python 3.2 > Python (Command Line)`. Si tout se passe bien, vous devriez obtenir une magnifique console qui reprend grosso modo ces informations (elles peuvent être différentes chez vous, ne vous inquiétez donc pas) :

Code : Console

```
Python 3.2 (r32:88445, Feb 20 2011, 21:29:02) [MSC v.1500 32 bit (Intel)] on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Qu'est-ce que c'est que ça ?

On verra plus tard. L'important c'est que vous ayez réussi à ouvrir la console d'interprétation de Python, le reste attendra le prochain chapitre.

Vous pouvez également passer par la ligne de commande Windows ; à cause des raccourcis je privilégie en général cette méthode, mais c'est une question de goût.
Allez dans le menu `Démarrer`, puis cliquez sur `Exécuter`. Dans la fenêtre qui s'affiche, tapez simplement « python » et la ligne de commande Python devrait s'afficher à nouveau. Sachez que vous pouvez directement vous rendre dans `Exécuter` en tapant le raccourci Windows + R.

Pour fermer l'interpréteur de commandes Python, vous pouvez tapez « exit() » puis appuyer sur la touche Entrée.

Sous Linux

En s'installant sur votre système, Python a créé un lien vers l'interpréteur sous la forme *python3.X* (le X étant le numéro de version installée).

Si par exemple vous avez installé Python 3.2, vous pouvez y accéder grâce à la commande :

Code : Console

```
$ python3.2
Python 3.2 (r32:88445, Mar  4 2011, 22:07:21)
[GCC 4.3.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Pour fermer la ligne de commande Python, n'utilisez pas CTRL + C mais CTRL + D (nous verrons plus tard pourquoi).

Sous Mac OS

Cherchez un dossier *Python* dans le dossier *Applications*. Pour lancer Python, ouvrez l'application `IDLE` de ce dossier. Vous êtes prêts à passer au concret !

Merci encore à Etienne-02 pour ses explications.
Allez, assez de théorie et d'installation comme ça, on commence la programmation dès le prochain chapitre, tout en douceur comme promis ! Et vu que vous avez été sages (et que je ne trouve aucune question pertinente à vous poser), je vous dispense de QCM. Vous vous en tirez donc à bon compte, profitez-en, la récréation s'achève ici ! 

Premiers pas avec l'interpréteur de commandes Python

Bien. Après les premières notions théoriques et l'installation de Python, il est temps de découvrir un peu l'interpréteur de commandes de ce langage. Si anodin que semblent ces petits tests, vous découvrirez dans cette partie les premiers rudiments de la syntaxe du langage et je vous conseille fortement de me suivre pas à pas, d'autant si vous êtes en face de votre premier langage de programmation.

Où est-ce qu'on est, là ?

Pour commencer, je vais vous demander de retourner dans l'interpréteur de commandes Python (je vous ai montré, en fonction de votre système d'exploitation, comment y accéder à la fin du chapitre précédent).

Je vous rappelle donc les informations que vous avez dans cette fenêtre, même si elles pourront être bien différentes chez vous en fonction de votre version et de votre système d'exploitation.

Code : Console

```
Python 3.2 (r32:88445, Feb 20 2011, 21:29:02) [MSC v.1500 32 bit (Intel)] on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Python vous souhaite à sa façon, la bienvenue dans son interpréteur de commandes.

Attends, attends, déjà, c'est quoi cet interpréteur ?

Vous vous souvenez (du moins je l'espère) dans le chapitre précédent, je vous ai donné une brève explication sur la différence entre langages compilés et langages interprétés. Et bien, cet interpréteur de commandes va nous permettre de tester directement du code. J'entre une ligne d'instruction, je fais entrée, je regarde ce que me dit Python (s'il me dit quelque chose), puis j'en entre une deuxième, une troisième... Cet interpréteur est particulièrement utile pour comprendre la base de Python et réaliser nos premiers petits programmes. Le principal inconvénient, en contre-partie, étant que le code que vous entrez n'est pas sauvegardé si vous ne le faites pas manuellement, mais chaque chose en son temps.

Dans la fenêtre que vous avez sous les yeux, l'information qui ne change pas d'un système d'exploitation à l'autre et qui est, somme toute, la plus importante, est la série des trois chevrons qui se trouvent en bas à gauche des informations ">>>". Ces trois signes signifient : je suis prêt à recevoir tes instructions.

Comme je l'ai dit, les langages de programmation respectent une syntaxe claire. On ne peut espérer que l'ordinateur comprenne si, dans cette fenêtre, vous commenciez à lui dire : j'aimerais que tu me codes un jeu vidéo génial  . Et autant que vous le sachiez tout de suite (bien qu'à mon avis vous vous en doutiez), on est très loin d'obtenir des résultats aussi spectaculaires à notre niveau.

Tout ça pour dire que si vous entrez n'importe quoi dans cette fenêtre, il est plus que probable que Python vous indique, clairement et fermement, qu'il n'a rien compris.

Si par exemple vous entrez "premier test avec Python", vous obtenez le résultat suivant :

Code : Python Console

```
>>> premier test avec Python
File "<stdin>", line 1
premier test avec Python
^
SyntaxError: invalid syntax
>>>
```

Et oui, l'interpréteur parle en anglais, et les instructions que vous entrerez, comme une écrasante majorité des langages de programmation, seront également en anglais. Mais pour l'instant, rien de bien compliqué : l'interpréteur vous indique qu'il a trouvé un problème dans votre ligne d'instruction. Il vous donne la ligne (en l'occurence la première) qu'il vous répète obligemment (ceci est très utile quand on travaille sur un programme de plusieurs centaines de lignes) puis vous dit ce qui l'arrête, ici : *SyntaxError: invalid syntax*. Limpide n'est-ce pas ? Ce que vous avez entré est incompréhensible pour Python. Enfin, la preuve qu'il n'est pas rancunier, c'est qu'il vous réaffiche une série de trois chevrons, montrant bien qu'il est prêt à retenter l'aventure  .

Bon, c'est bien joli de recevoir un message d'erreur au premier test, mais je me doute que vous aimeriez bien voir des trucs qui marchent, maintenant. C'est parti donc  .

Vos premières instructions : un peu de calcul mental pour l'ordinateur

C'est assez trivial, quand on y pense, mais je pense qu'il s'agit d'une excellente manière d'aborder pas à pas la syntaxe de Python. Nous allons donc essayer d'obtenir des résultats à des calculs, plus ou moins compliqués. Je vous rappelle encore une fois qu'exécuter les tests en même temps que moi sur votre machine est une très bonne façon de vous rendre compte de la syntaxe et surtout, de la retenir.

Entrer un nombre

Vous avez pu voir, sur notre premier (et à ce jour notre dernier) test que Python n'aimait pas particulièrement les suites de lettres qu'il ne comprend pas. Cependant, l'interpréteur adore les nombres. D'ailleurs, il les accepte sans sourciller, sans une seule erreur :

Code : Python Console

```
>>> 7
7
>>>
```

D'accord, ce n'est pas extraordinaire. On entre un nombre, et l'interpréteur retourne ce nombre. Mais dans bien des cas, ce simple retour indique qu'il a bien compris et que ce que vous avez entré est en accord avec sa syntaxe. Vous pouvez de même entrer des nombres à virgule :

Code : Python Console

```
>>> 9.5
9.5
>>>
```

Attention : on utilise ici la notation anglo-saxonne, le point remplace la virgule. La virgule a un tout autre sens pour Python, prenez donc cette habitude dès maintenant.

Les nombres à virgule se comportent parfois étrangement, c'est pourquoi il faut les manier avec prudence.

Code : Python Console

```
>>> 8.13
8.130000000000008
>>>
```

Euh... pourquoi ce résultat approximatif ?

Python n'y est pas pour grand chose. En fait, le problème vient en grande partie de la façon dont les nombres à virgule sont écrits dans la mémoire de votre ordinateur. C'est pourquoi en programmation, quand on le peut, on préfère travailler avec des nombres entiers. Cependant, vous remarquerez que l'erreur est infime et qu'elle n'a pas d'impact formidable sur les calculs. Les applications qui ont besoin d'une précision mathématique à toute épreuve essayent de corriger ces défauts par d'autres moyens, mais ici ce ne sera pas nécessaire .

Il va de soi que l'on peut tout aussi bien rentrer des nombres négatifs (vous pouvez faire l'essai d'ailleurs).

Opérations courantes

Bon, il est temps d'apprendre à utiliser les principaux opérateurs de Python qui vont vous servir pour la grande majorité de vos programmes.

Addition, soustraction, multiplication, division

Pour effectuer ces opérations, on utilise respectivement les symboles "+", "-", "*" et "/".

Code : Python Console

```
>>> 3+4
7
>>> -2+93
91
>>> 9.5+2
11.5
>>>
```

Faites également des tests pour la soustraction, la multiplication et la division, il n'y a rien de difficile.

Division entière et modulo

Si vous avez pris le temps de tester la division, vous vous êtes rendu compte que le résultat est donné en virgule flottante.

Code : Python Console

```
>>> 10/5
2.0
>>> 10/3
3.3333333333333335
>>>
```

Il existe deux autres opérateurs qui permettent de connaître le résultat d'une division entière et le reste de cette division.

Le premier opérateur est le signe "//". Il permet d'obtenir la partie entière d'une division.

Code : Python Console

```
>>> 10//3
3
>>>
```

L'opérateur "%" que l'on appelle "modulo" permet de connaître le reste de la division :

Code : Python Console

```
>>> 10%3
1
>>>
```

Cette notion de partie entière et de reste de division n'est pas bien difficile à comprendre et vous servira très probablement par la suite.

Si vous avez du mal à en saisir le sens, sachez donc que :

- La partie entière de la division de 10 par 3 est le résultat de cette division, sans tenir compte des chiffres au-delà de la virgule (en l'occurence, 3).
- Pour obtenir le reste de 10/3, on regarde quelle est la partie entière. Ici elle est de 3 (`10//3 `). On multiplie cette partie entière par 3 (celui de 10/3) et on obtient... 9  . Entre 9 et 10, on a 1... et c'est le reste de notre division. Vous devriez retrouver ces chiffres facilement en posant la division sur papier.

Souvenez-vous bien de ces deux opérateurs, et surtout le modulo "%" dont vous aurez besoin dans vos programmes futurs.
Voilà ! Ce premier tour d'horizon s'est bien passé pour vous je l'espère, il n'y avait rien de vraiment compliqué. Mais finalement, on n'est pas vraiment rentré dans le sujet de la syntaxe, et ce qu'on a fait, une calculette de poche peut le faire aussi bien, inutile d'apprendre à programmer pour savoir faire 3+8. Le chapitre suivant va vous faire découvrir le monde des variables, et vous aurez vos premiers concepts spécifiques à Python à ingurgiter.
N'hésitez pas, avant de passer à ce chapitre, à tester quelques opérations, notamment des plus complexes, avec des parenthèses pour tester par exemple la priorité des opérateurs.

Le monde merveilleux des variables

Dans le chapitre précédent, vous avez entré vos premières instructions en langage Python, bien que vous ne vous en soyez peut-être pas rendu compte. Il est également vrai que les instructions entrées auraient marché dans la plupart des langages. Ici, cependant, nous commençons à approfondir un petit peu la syntaxe du langage, tout en découvrant un concept important de programmation : les variables.

Ce concept est essentiel et vous ne pouvez absolument pas faire de croix dessus, mais je vous rassure il n'y a rien de compliqué, que de l'utile et de l'agréable .

C'est quoi, des variables ? Et à quoi ça sert ?

Les variables sont l'un des concepts qui se retrouvent dans la majorité (si ce n'est la totalité) des langages de programmation. Autant dire que sans variable, on ne peut pas programmer, sans exagération.

C'est quoi, des variables ?

Une variable est une donnée de votre programme. C'est un code alpha-numérique que vous allez lier à une donnée de votre programme, pour pouvoir l'utiliser à plusieurs reprises et faire des calculs un peu plus intéressants dessus. C'est bien joli de savoir faire des opérations, mais si on ne peut pas même stocker le résultat, ça devient très vite ennuyeux.

Comment ça marche ?

Le plus simplement du monde. Vous allez dire à Python : je veux que dans une variable que je nomme *age* tu stockes mon âge, pour que je puisse le retenir (si j'ai la mémoire très courte), l'augmenter (à mon anniversaire) et l'afficher si besoin est.

Comme je vous l'ai dit, on ne peut passer à côté des variables. Vous ne voyez peut-être pas encore tout l'intérêt de stocker des informations de votre programme et pourtant, si vous ne stockez rien, vous ne pouvez pratiquement rien faire.

En Python, pour donner une valeur à une variable, il suffit d'écrire `nom_de_la_variable = valeur` .

Une variable respecte quelques règles de syntaxe incontournables :

1. Le nom de la variable n'est composé que de lettres, majuscules ou minuscules, de chiffres et du symbole souligné (underscore en anglais) "_". Les caractères accentués ne font pas partie des lettres autorisées.
2. Le nom de la variable ne peut commencer par un chiffre.
3. Le langage Python est sensible à la casse, ce qui signifie que des lettres majuscules et minuscules ne constituent pas la même variable (*AGE* est différent de *aGe*, elle-même différente de *age*).

Au-delà de ces règles de syntaxe incontournables puisque parties intégrantes du langage, il existe des conventions définies par les programmeurs. L'une d'elle, que j'ai tendance à utiliser le plus souvent, consiste à écrire la variable en minuscule et de remplacer les espaces éventuels par un signe souligné "_". Si je dois créer une variable contenant mon âge, elle se nommera donc *mon_age*. Une autre généralement utilisée est de remplacer les caractères commençant chaque mot par la majuscule de ce caractère, à l'exception du premier mot constituant la variable. Donc, la variable contenant mon âge se nommera donc *monAge*.

Vous pouvez utiliser la convention qui vous plaît, même en créer une bien à vous, mais essayez de rester cohérent et de n'utiliser qu'une seule convention d'écriture, car vous repérer dans vos variables est essentiel dès lors qu'on commence à travailler sur des programmes un peu plus consistants.

Ainsi, si je veux associer mon âge à une variable, la syntaxe sera :

Code : Python

```
mon_age = 21
```

L'interpréteur vous remet aussitôt trois chevrons sans aucun message. Cela signifie qu'il a bien compris et qu'il n'y a eu aucune erreur.

Sachez qu'on appelle cette étape **l'affectation de variable**. On dit en effet qu'on a affecté 21 à la variable *mon_age*.

On peut afficher la valeur de cette variable en l'entrant simplement dans l'interpréteur de commandes.

Code : Python Console

```
>>> mon_age
21
>>>
```

N.B.: Les espaces séparant = du nom et de la valeur de la variable sont facultatifs. Je les mets pour des raisons de lisibilité, mais il est possible, par la suite, que je ne le fasse pas, ne soyez donc pas surpris.

Bon, c'est bien joli tout ça, mais qu'est-ce qu'on fait avec cette variable ?

Et bien, tout ce que vous avez déjà fait dans le chapitre précédent, en utilisant cette variable comme un nombre à part entière. Vous pouvez même affecter d'autres variables à partir de calculs sur la première, et c'est là toute la puissance de ce mécanisme.

Essayons par exemple d'augmenter de 2 la variable mon_age :

Code : Python Console

```
>>> mon_age = mon_age + 2
>>> mon_age
23
>>>
```

Encore une fois, lors de l'affectation de la variable, rien ne s'affiche, ce qui est parfaitement normal.

Maintenant, essayons d'affecter une autre variable d'après la valeur de *mon_age*.

Code : Python Console

```
>>> mon_age_x2 = mon_age * 2
>>> mon_age_x2
46
>>>
```

Encore une fois, je vous invite à tester en long, en large et en travers cette possibilité. Le concept n'est pas compliqué, mais extrêmement puissant. De plus, comparé à certains langages, affecter une valeur à une variable est des plus simples. Si la variable n'est pas créée, Python le fait automatiquement. Si la variable existe déjà, l'ancienne valeur est supprimée et remplacée par la nouvelle. Quoi de plus simple ?

Certains mot-clés de Python sont réservés, c'est-à-dire que vous ne pouvez pas créer des variables portant ce nom.

En voici la liste pour Python 3.

and	del	from	none	true
as	elif	global	nonlocal	try
assert	else	if	not	while
break	except	import	or	with

class	false	in	pass	yield
continue	finally	is	raise	
def	for	lambda	return	

Ces mot-clés sont utilisés par Python, vous ne pouvez pas construire des variables avec. Vous allez découvrir dans la suite de ce cours la majorité de ces mot-clés et comment ils s'utilisent .

Les types de données en Python

Là se trouve un concept très important que l'on retrouve dans beaucoup de langages de programmation. Ouvrez bien en grand vos oreilles, ou plutôt vos yeux, car vous devrez être parfaitement à l'aise avec ce concept pour continuer ce tutoriel. Rassurez-vous toutefois, du moment que vous êtes attentifs, il n'y a rien de délicat à comprendre.

Qu'entend-t-on par "type de donnée" ?

Jusqu'ici, vous n'avez travaillé qu'avec des nombres. Et s'il faut bien avouer qu'on ne fera que très rarement un programme sans aucun nombre, c'est loin d'être la seule donnée que l'on peut utiliser en Python. A terme, vous serez même capable de créer vos propres types de données, mais n'anticipons pas.

Python a besoin de savoir quels types de données sont utilisées, d'abord, et tout bêtement, pour savoir quelles opérations il peut effectuer. Vous allez dans ce chapitre apprendre à faire des chaînes de caractères, et multiplier une chaîne de caractères ne se fait pas du tout comme multiplier un nombre. Pour certains types de données, la multiplication n'a d'ailleurs aucun sens. Python associe donc à chaque donnée un type qui va définir les opérations qu'il peut faire sur cette donnée en particulier.

Les différents types de données

Nous n'allons voir ici que les incontournables et les plus faciles à manier. Des chapitres entiers seront consacrés aux types plus complexes.

Les nombres

Les entiers

Et oui, Python différencie les entiers des nombres à virgule flottante !

Pourquoi cela ?

Initialement, c'est surtout pour une question de place en mémoire, mais les opérations que l'on effectue sur des nombres à virgule ne sont pour un ordinateur pas les mêmes que les opérations que l'on effectue sur un entier, et cette distinction reste encore d'actualité de nos jours.

Le type entier se nomme *int* en Python ("entier" en anglais). La syntaxe d'un entier est un nombre sans virgule.

Code : Python

```
3
```

Nous avons vu dans le chapitre précédent les opérations que l'on pouvait effectuer sur ce type de données, et quand bien même vous ne vous en souviendriez pas, les deviner est assez élémentaire.

Les flottants

Les flottants sont les nombres à virgule. Ils se nomment *float* en Python ("flottant" en anglais). La syntaxe d'un nombre flottant est celle d'un nombre à virgule (n'oubliez pas de remplacer la virgule par un point). Si ce nombre n'a pas de partie flottante mais que vous voulez qu'il soit considéré par le système comme un flottant, vous pouvez lui mettre une partie flottante de 0 (exemple *52.0*).

Code : Python

```
3.152
```

Les nombres après la virgule ne sont pas infinis, puisque rien n'est infini en informatique. Mais la précision est assez importante pour travailler sur des données très précises.

Les chaînes de caractères

Heureusement, Python ne permet pas seulement de traiter des nombres, bien loin de là. Le dernier type "simple" que nous verrons dans ce chapitre consiste en les chaînes de caractères. Elles permettent de stocker une série de lettres, une phrase, pourquoi pas.

On peut écrire une chaîne de caractères de différentes façons :

- Entre guillemets (`"ceci est une chaîne de caractères"`)
- Entre apostrophes (`'ceci est une chaîne de caractères'`)
- Entre triples guillemets (`"""ceci est une chaîne de caractères"""`)

Quelques remarques sont à faire quant aux chaînes de caractères :

- On peut, à l'instar des nombres (et de tous les types de données) stocker une chaîne de caractères dans une variable (`ma_chaine = "Bonjour, la foule !"`)
- Si vous utilisez les délimiteurs simples (le guillemet ou l'apostrophe) pour encadrer une chaîne de caractères, il faut échapper, au choix, les guillemets ou les apostrophes que vous contiendrez dans la chaîne. Par exemple, si vous tapez :

Code : Python

```
chaine = 'je m'appelle Prolixe.'
```

Vous obtenez le message :

Code : Console

```
File "<stdin>", line 1
chaine = 'je m'appelle Prolixe.'
^
SyntaxError: invalid syntax
```

Ceci est dû au fait que l'apostrophe de "m'appelle" est considéré par Python comme la fin de la chaîne, et qu'il ne sait pas quoi faire de tout ce qui se trouve au-delà.

Pour pallier ce problème, il faut **échapper** les apostrophes se trouvant au coeur de la chaîne (si les délimiteurs sont des apostrophes, il va de soi que si vous utilisez des guillemets, le problème ne se pose pas avec des apostrophes). Pour se faire, on insère l'anti-slash "\" avant les apostrophes contenus dans le message.

Code : Python

```
chaine = 'je m\'appelle Prolixe.'
```

On doit également échapper les guillemets si on utilise les guillemets comme délimiteurs :

Code : Python

```
chaine2 = "\"Le seul individu formé, c'est celui qui a appris
comment apprendre (...)\" (Karl Rogers, 1976)"
```

- Le caractère d'échappement "\" est utilisé pour symboliser d'autres signes très utiles. Ainsi, "\n" symbolise un saut de ligne (`"essai\nsur\nplusieurs\nlignes"`). Pour écrire un véritable anti-slash dans une chaîne, il faut lui-même l'échapper (donc écrire "\\").

  N.B.: l'interpréteur affiche les sauts de lignes comme on les entre, c'est-à-dire sous forme de "\n". Nous verrons dans la partie qui vient juste après comment afficher réellement ces chaînes de caractères, et pourquoi l'interpréteur ne les affiche pas comme il le devrait.
- Utiliser les triple guillemets pour encadrer une chaîne de caractères dispense d'échapper les guillemets, les apostrophes, et permet d'écrire plusieurs lignes sans symboliser les retours à la ligne grâce à "\n".

Code : Python Console

```
>>> chaine3 = """Ceci est un nouvel
... essai sur plusieurs
... lignes"""
>>>
```

Notez que les trois chevrons sont remplacés par trois points, ce qui signifie que l'interpréteur considère que vous n'avez pas fini d'écrire cette instruction, qui ne s'achève en effet que quand vous refermez la chaîne avec trois nouveaux guillemets. Les sauts de lignes seront remplacés automatiquement par des "\n" dans la chaîne.

Vous pouvez utiliser, à la place des trois guillemets, trois apostrophes qui ont exactement le même but. Je n'utilise personnellement pas ces délimiteurs, mais sachez qu'ils existent et ne soyez pas surpris si vous les voyez un jour dans un code source.

Voilà, le rapide tour d'horizon des types simples est accompli. Qualifier les chaînes de caractères de type simple n'est pas strictement vrai, mais nous n'allons pas dans ce chapitre rentrer dans le détail des opérations que l'on peut effectuer sur ces chaînes, c'est inutile pour l'instant, et ce serait hors sujet. Cependant, rien ne vous empêche de tester vous même quelques opérations comme l'addition et la multiplication (dans le pire des cas, Python vous dira qu'il ne peut pas faire ce que vous lui demandez), et comme nous l'avons vu, il est peu rancunier  .

Un petit bonus

Dans le chapitre précédent, nous avons vu les opérateurs "classiques" pour manipuler des nombres, mais aussi, comme on le verra plus tard, d'autres types de données. D'autres opérateurs ont été créés afin de simplifier la manipulation des variables.

Vous serez amené par la suite, et assez régulièrement, à incrémenter des variables. L'incrémentation désigne l'augmentation de la valeur d'une variable d'un certain nombre. Jusqu'ici, j'ai procédé comme ceci pour augmenter une variable de 1 :

Code : Python

```
variable = variable + 1
```

Cette syntaxe est claire, intuitive, mais assez longue, et les programmeurs, tout le monde le sait, sont des fainéants nés ;). On a donc trouvé plus court :

Code : Python

```
variable += 1
```

L'opérateur += revient à ajouter la valeur précisée à sa suite à la variable qui le précède. Les opérateurs -=, *= et /= existent également, bien qu'ils soient moins utilisés.

Quelques trucs et astuces pour vous faciliter la vie

Python propose un moyen simple de permuter deux variables (échanger leur valeur). Dans d'autres langages, il est nécessaire de passer par une troisième variable qui retient l'une des deux valeurs... ici c'est bien plus simple :

Code : Python Console

```
>>> a = 5
>>> b = 32
>>> a,b = b,a # permutation
>>> a
32
>>> b
5
>>>
```

Comme vous le voyez, après l'exécution de la ligne 3, les variables *a* et *b* ont échangé leur valeur. On retrouvera cette distribution d'affectation bien plus loin  .

On peut également affecter une valeur à plusieurs variables assez simplement :

Code : Python Console

```
>>> x = y = 3
>>> x
3
>>> y
3
>>>
```

Enfin, ce n'est pas encore d'actualité pour vous, mais sachez qu'on peut couper une instruction Python, pour l'écrire sur deux lignes ou plus.

Code : Python Console

```
>>> 1 + 4 - 3 * 19 + 33 - 45 * 2 + (8 - 3) \
... -6 + 23.5
-86.5
>>>
```

Comme vous le voyez, le symbole "\" permet, avant un saut de ligne, de dire à Python "cette instruction se poursuit en-dessous". Vous pouvez ainsi morceler votre instruction sur plusieurs lignes.

Première utilisation des fonctions

Et bien, ça avance gentiment, et je me permets d'introduire l'utilisation des fonctions dans le chapitre des variables, même s'il s'agit, en fait, bien plus d'une application concrète de ce que vous avez appris juste à l'instant. Un chapitre entier sera consacré aux fonctions, mais utiliser celles que je vais vous montrer n'est pas sorcier et pourra vous être utile.

Utiliser une fonction

A quoi servent les fonctions ?

Une fonction exécute un certain nombre d'instructions déjà enregistrées. En gros, c'est comme si vous enregistriez un groupe d'instructions pour faire une action précise et que vous lui donniez un nom. Vous n'avez plus ensuite qu'à appeler cette fonction par son nom autant de fois que nécessaire (cela évite bon nombre de répétitions). Toutefois, il est encore bien trop tôt pour apprendre à créer vos fonctions.

La plupart des fonctions ont besoin d'au moins un paramètre pour travailler sur une donnée. Les fonctions que je vais vous montrer ne font pas exception. Ce concept vous semble peut-être un peu difficile à saisir dans son ensemble, mais rassurez-vous, les exemples devraient rendre tout limpide.

Les fonctions s'utilisent en suivant cette syntaxe : *nom_de_la_fonction(parametre_1,parametre_2,...,parametre_n)*.

- Vous commencez par écrire le nom de la fonction.
- Vous placez entre parenthèses les paramètres de la fonction. Si la fonction n'attend aucun paramètre, vous devrez quand même mettre les parenthèses, sans rien entre elles.

La fonction "type"

Dans la partie précédente, je vous ai présenté les types de données simples, du moins une partie d'entre eux. Une des grandes puissances de Python est qu'il comprend automatiquement de quel type est une variable et cela lors de son affectation. Mais il est pratique de pouvoir savoir de quel type est une variable.

La syntaxe de cette fonction est simple :

Code : Python

```
type(nom_de_la_variable)
```

La fonction retourne le type de la variable passée en paramètre. Vu que nous sommes dans l'interpréteur de commandes, cette valeur sera affichée. Si vous entrez dans l'interpréteur les lignes suivantes :

Code : Python

```
a = 3
type(a)
```

Vous obtenez :

Code : Python Console

```
<class 'int'>
```

Python vous indique donc que la variable a appartient à la classe des entiers. Ce mot de classe ne sera pas approfondi avant bien des chapitres, mais sachez qu'on peut le rapprocher d'un type de donnée.

Vous pouvez faire le test sans passer par des variables :

Code : Python Console

```
>>> type(3.4)
<class 'float'>
>>> type("un essai")
<class 'str'>
>>>
```

N.B.: str est l'abréviation de "string" qui signifie chaîne (sous-entendu, de caractère) en anglais.

Vous serez peut-être amené à utiliser cette fonction pour différencier des actions en fonction de types de données, mais pour l'instant il est juste utile de savoir à quel type de donnée appartient une variable.

La fonction "print"

La fonction *print* permet d'afficher une ou plusieurs variables.

Mais... on ne fait pas exactement la même chose en entrant juste le nom de la variable dans l'interpréteur ?

Oui, et non. L'interpréteur l'affiche, car il affiche automatiquement tout ce qu'il peut, pour pouvoir suivre les étapes d'un programme. Cependant, quand vous ne travaillerez plus avec l'interpréteur, taper simplement le nom de la variable n'aura aucun effet. De plus, et vous l'aurez sans doute remarqué, l'interpréteur entoure les chaînes de caractères de délimiteurs et affiche les caractères d'échappement, tout ceci encore pour des raisons de clarté.

La fonction *print* est dédiée à l'affichage uniquement. Son nombre de paramètres est variable, c'est à dire que vous pouvez lui demander d'afficher une ou plusieurs variables. Considérez cet exemple :

Code : Python

```
a = 3
print(a)
a = a + 3
b = a - 2
print("a =", a, "et b =", b)
```

Le premier **appel** à *print* se contente d'afficher la variable a, c'est à dire *3*.
Le second appel à *print* affiche :

Code : Python Console

```
a = 6 et b = 4
```

Ce deuxième appel à *print* est peut-être un peu plus dur à comprendre. En fait, on passe quatre paramètres à print, deux chaînes de caractères, et a et b. Quand Python interprète cette fonction, il va afficher dans l'ordre de passage les paramètres en les séparant par un espace.

Relisez bien cet exemple, il montre tout l'intérêt des fonctions. Si vous avez du mal à le comprendre dans son ensemble, décortiquez-le en prenant chaque paramètre indépendamment.

Testez l'utilisation de *print* avec d'autres types de données, et en insérant des chaînes avec des sauts de lignes et des caractères échappés, pour bien vous rendre compte de la différence.

Un petit Hello World ?

Quand on fait un cours sur un langage quelconque, il est pour ainsi dire de tradition de présenter le programme Hello World qui permet de montrer assez rapidement la syntaxe superficielle d'un langage.

Le but du jeu est très simple : faire un programme qui affiche *Hello World*. Dans certains langages, notamment les langages compilés, vous pourrez avoir autour d'une dizaine de lignes pour obtenir ce résultat. En Python, comme nous venons de le voir, il ne suffit que d'une seule ligne :

Code : Python

```
print("Hello World !")
```

Pour ma part, je cherche toujours quelque chose de plus simple  .

Pour plus d'informations, n'hésitez pas à consulter la page Wikipédia consacrée au programme Hello World, vous avez même le code de ce programme sous un grand nombre de langages, ce peut être intéressant  .

Et voilà, cette fois je crois que tout est dit. Vous n'avez peut-être pas l'impression d'avoir appris grand chose d'utile, et ce du fait de la facilité relative des concepts, mais à chaque chapitre, nous entrons un peu plus avant dans la syntaxe du langage, et dès le prochain vous pourrez commencer à faire des choses réellement intéressantes.

Les structures conditionnelles

Jusqu'à présent, nous avons testé des instructions d'une façon linéaire : l'interpréteur exécutait au fur et à mesure le code que vous entriez dans la console. Mais nos programmes seraient bien pauvres si l'on ne pouvait de temps à autre demander à exécuter certaines instructions dans un cas, et d'autres instructions dans un autre cas. Et le programmeur inventa... la condition .

Vos premières conditions et blocs d'instructions

Forme minimale en if

Les conditions sont un concept essentiel en programmation (oui oui, je me répète à force, mais il faut avouer que des concepts essentiels, on n'a pas fini d'en voir). Elles vont vous permettre de faire une action précise si, par exemple, une variable est positive, et une autre action si cette variable est négative, et encore une troisième si la variable est nulle. Comme un bon exemple vaut mieux que plusieurs lignes d'explications, voici un exemple clair d'une condition prise sous sa forme la plus simple.

N.B.: Dès à présent dans mes exemples, j'utiliserais des commentaires. Les commentaires sont des messages qui sont ignorés par l'interpréteur et qui permettent de donner des indications sur le code (car, vous vous en rendrez compte, relire ses programmes après plusieurs semaines d'abandon, sans commentaire, ce peut être parfois plus qu'ardu). En Python, un commentaire débute par un dièse (#) et se termine par un saut de ligne. Tout ce qui est compris entre ce # et ce saut de ligne est ignoré. Tous les commentaires sont donc soit sur toute une ligne (on place le "#" en début de ligne) soit après une instruction (on place le # après la ligne de code pour la commenter plus spécifiquement).

Cela étant posé, revenons à nos conditions :

Code : Python Console

```
>>> # premier exemple de condition
>>> a = 5
>>> if a > 0: # si a est supérieur à 0
...     print("a est supérieur à 0.")
...
a est supérieur à 0.
>>>
```

Détaillons ce code, ligne par ligne :

1. La première ligne est un commentaire décrivant qu'il s'agit du premier test de condition. Elle est ignorée par l'interpréteur et sert juste à vous renseigner sur le code qui va suivre.
2. Cette ligne, vous devriez la comprendre sans aucune aide. On se contente d'affecter 5 à la variable *a*.
3. Ici se trouve notre test conditionnel. Il se compose, dans l'ordre :
   - Du mot clé *if* qui signifie "si" en anglais
   - De la condition proprement dite *a > 0* qu'il est facile de lire. Une liste des opérateurs de comparaison possibles sera présentée plus bas.
   - Du signe deux points (:) qui termine la condition. Ce signe est indispensable : Python affichera une erreur de syntaxe si vous l'omettez.
4. Ici se trouve l'instruction à exécuter dans le cas où a est supérieur à 0. Après l'appui sur entrée à la ligne précédente, l'interpréteur vous présente la série de trois points qui signifie qu'il attend que vous entriez le **bloc d'instructions** concerné avant de l'interpréter. Cette instruction (et les autres instructions à exécuter s'il y en a) est indentée, c'est-à-dire décalée sur la droite. Des explications supplémentaires seront données sur les indentations un peu plus bas
5. L'interpréteur vous ré-affiche trois nouveaux points, si vous voulez entrer une nouvelle instruction dans ce bloc d'instructions. Ce n'est pour l'instant pas le cas. Vous appuyez donc sur entrée sans avoir rien écrit et l'interpréteur vous affiche le message "a est supérieur à 0", ce qui est assez logique, vu que a est effectivement supérieur à 0 .

Il y a deux notions importantes sur lesquelles je dois à présent revenir, elles sont complémentaires ne vous en faites pas.

La première est celle de bloc d'instructions. On entend par bloc d'instructions une série d'instructions qui s'exécutent dans un certain cas précis (par condition, comme on vient de le voir, par répétition, comme on le verra plus tard...). Ici, notre bloc d'instructions n'est constitué que d'une seule instruction (la ligne 4 faisant appel à print). Mais rien ne vous empêche de mettre

plusieurs instructions dans ce bloc.

Code : Python

```
a = 5
b = 8
if a > 0:
    # on incrémente la valeur de b
    b += 1
    # on affiche les valeurs des variables
    print("a =",a,"et b =",b)
```

La seconde notion importante est celle d'indentation. On entend par indentation un certain décalage vers la droite obtenu, au choix, par un (ou plusieurs) espaces ou tabulations.

J'avais tendance, jusqu'à peu, à utiliser (et conseiller) les tabulations pour marquer l'indentation. Cependant, si on s'appuie sur la PEP 08 (les PEPs sont les **Python Enhancement Proposals** : propositions d'amélioration de Python), il est plus judicieux d'utiliser 4 espaces par niveau d'indentation. Je respecterais donc dorénavant cette convention.

Les indentations sont essentielles pour Python. Il ne s'agit pas, comme dans d'autres langages tels que le C++ ou le Java d'un confort de lecture, mais bien d'un moyen pour l'interpréteur de savoir où se trouve le début et la fin d'un bloc.

Forme complète (if, elif et else)

Les limites de la condition simple en if

La première forme de condition que l'on vient de voir est pratique, mais assez incomplète.

Si l'on considère par exemple une variable a de type entier. On souhaite faire une action si elle est positive, et une autre différente si elle est négative. Il est possible d'obtenir ce résultat avec la forme simple d'une condition :

Code : Python

```
a = 5
if a > 0: # si a est positif
    print("a est positif.")
if a < 0: # a est négatif
    print("a est négatif.")
```

Amusez-vous à changer la valeur de a, et ré-exécutez les conditions, vous obtiendrez des messages différents, sauf si a est égal à 0. En effet, aucune action n'a été prévue si a est égal à 0.

Cette méthode n'est pas optimale. Tout d'abord parce qu'elle nous oblige à faire deux conditions séparées pour tester une même variable. De plus, et même si c'est dur à concevoir par cet exemple, dans le cas où la variable remplirait les deux conditions (ici c'est impossible bien entendu), les deux portions de code s'exécuteraient.

La condition "if" est donc bien pratique, mais insuffisante.

L'instruction else:

Le mot-clé *else* qui signifie "sinon" en anglais, permet de définir une première forme de complément à notre instruction *if*.

Code : Python

```
age = 21
if age >= 18: # si age est supérieur ou égal à 18
    print("Vous êtes majeur.")
else: # sinon (age inférieur à 18)
    print("Vous êtes mineur.")
```

Je pense que cet exemple suffit amplement à exposer l'utilisation de *else*. La seule "subtilité" est de bien se rendre compte que Python exécute soit l'un, soit l'autre, et jamais les deux. Notez que cette instruction doit se trouver au même niveau d'indentation que l'instruction *if* qu'elle complète. De plus, elle se termine également par deux points puisqu'il s'agit d'une condition, même si elle est sous-entendue.

L'exemple de tout à l'heure pourrait donc se présenter comme suit, avec l'utilisation de else :

Code : Python

```
a = 5
if a > 0:
    print("a est supérieur à 0.")
else:
    print("a est inférieur ou égal à 0.")
```

Mais... le résultat n'est pas tout à fait le même, si ?

Non, en effet. Vous vous rendrez compte que cette fois, le cas où a égale 0 est bien pris en compte. En effet, la condition initiale prévoit d'exécuter le premier bloc d'instructions si a est strictement supérieur à 0. Sinon, on exécute le second bloc d'instructions.

Si l'on veut faire la différence entre les nombres positifs, négatifs ou nuls, il va falloir utiliser une condition intermédiaire.

L'instruction elif:

Le mot clé *elif* est une contraction de "else if", que l'on peut traduire très littéralement par "sinon si". Dans l'exemple que nous venons juste de voir, l'idéal serait de faire :

- Si *a* est strictement supérieur à 0, on dit qu'il est positif
- Sinon si *a* est strictement inférieur à 0, on dit qu'il est négatif
- Sinon (*a* ne peut qu'être égal à 0), *a* est nul.

Traduit en langage Python, ça donne :

Code : Python

```
if a > 0: # positif
    print("a est positif.")
elif a < 0: # négatif
    print("a est négatif.")
else: # nul
    print("a est nul.")
```

De même que le *else*, le *elif* est sur le même niveau d'indentation que le *if* de départ. Il se termine de même par deux points, cependant, entre le *elif* et les deux points se trouve une nouvelle condition. Linéairement, le schéma d'exécution se traduit comme suit :

1. On regarde si *a* est strictement supérieur à 0. Si c'est le cas, on affiche "a est positif" et on s'arrête là
2. Sinon, on regarde si *a* est strictement inférieur à 0. Si c'est le cas, on affiche "a est négatif" et on s'arrête
3. Sinon, on affiche "a est nul".

Attention : quand je dis "on s'arrête", il va de soi que c'est uniquement pour cette condition. Si il y a du code après les trois blocs d'instructions, il sera exécuté dans tous les cas.

Vous pouvez mettre autant de *elif* que vous voulez après une condition en *if*. Tout comme le *else*, cette instruction est facultative, et, quand bien même vous construiriez une instruction en *if*, *elif*, vous n'êtes pas du tout obligé de prévoir un *else* après. En revanche, l'instruction else ne doit être mise qu'une fois au maximum, clôturant le bloc de la condition. Deux instructions *else* dans une même condition ne sont pas envisageables et n'auraient de toute façon pas grand sens.

Sachez qu'il est heureusement possible d'imbriquer des conditions, et, dans ce cas, l'indentation permet de comprendre clairement le schéma d'exécution du programme. Je vous laisse essayer cette possibilité, je ne vais pas tout faire à votre place, non plus  .

De nouveaux opérateurs

Les opérateurs de comparaison

Les conditions doivent nécessairement introduire de nouveaux opérateurs, dits **opérateurs de comparaison**. Je vais les présenter très brièvement, vous laissant l'initiative de faire des tests, car ils ne sont réellement pas difficiles à comprendre.

Opérateur	Signification littérale
<	Strictement inférieur à
>	Strictement supérieur à
<=	Inférieur ou égal à
>=	Supérieur ou égal à
==	Egal à
!=	Différent de

Attention : l'égalité de deux valeurs est comparée avec l'opérateur "==" et non "=". Ce dernier est en effet l'opérateur d'affectation et ne doit pas être utilisé dans une condition.

Prédicats et booléens

Avant d'aller plus loin, sachez que vos conditions se trouvant par exemple entre "if" et les deux points sont appelés des **prédicats**. Vous pouvez tester ces prédicats directement dans l'interpréteur, pour comprendre les explications qui vont suivre.

Code : Python Console

```
>>> a = 0
>>> a == 5
False
>>> a > -8
True
>>> a != 33.19
True
>>>
```

L'interpréteur renvoie tantôt *True* (c'est-à-dire "vrai") ou tantôt *False* (c'est-à-dire "faux").

True et *False* sont les deux valeurs possibles d'un type que nous n'avons pas vu jusqu'ici : le type booléen (bool).

N'oubliez pas que *True* et *False* sont des valeurs ayant leur première lettre en majuscule. Si vous commencez à écrire *true* sans un 'T' majuscule, Python ne va pas comprendre.

Les variables de ce type ne peuvent prendre comme valeur que vrai ou faux et peuvent être pratiques, justement, pour stocker des prédicats, de la façon dont nous l'avons vu ou d'une façon plus détournée.

Code : Python Console

```
>>> age = 21
>>> majeur = False
>>> if age >= 18:
>>>     majeur = True
>>>
```

A la fin de cet exemple, majeur vaut "True", c'est-à-dire "vrai", si age est supérieur ou égal à 18. Sinon, il continue de valoir "False". Les booléens ne vous semblent pas très utiles pour l'instant je pense, mais vous verrez qu'ils rendent de grands services.

Les mots clés "and", "or" et "not"

Il arrive souvent que nos conditions doivent tester plusieurs prédicats. Par exemple, si l'on veut savoir si une variable quelconque, de type entier, se trouve dans un intervalle précis (c'est-à-dire comprise entre deux nombres). Avec nos méthodes actuelles, le plus simple serait de faire comme suit :

Code : Python

```
# on fait un test pour savoir si a est comprise dans l'intervalle
allant de 2 à 8 inclus
a = 5
if a >= 2:
    if a <= 8:
        print("a est dans l'intervalle.")
    else:
        print("a n'est pas dans l'intervalle.")
else:
    print("a n'est pas dans l'intervalle.")
```

Ça marche, mais c'est assez lourd. D'autant que pour être sûr qu'un message soit affiché à chaque fois, il faut fermer d'un *else* les deux conditions (la seconde étant imbriquée dans la première). Si vous avez du mal à comprendre cet exemple, prenez le temps de le décortiquer, ligne par ligne, il n'y a rien que de très simple.

Il existe cependant le mot clé *and* ("et" en anglais) qui ici va nous rendre un fier service. En effet, ce qu'on veut tester, c'est si *a* est supérieur ou égal à 2 et inférieur ou égal à 8. L'on peut donc réduire ainsi nos conditions imbriquées :

Code : Python

```
if a>=2 and a<=8:
    print("a est dans l'intervalle.")
else:
    print("a n'est pas dans l'intervalle.")
```

Simple et bien plus compréhensible, avouez-le  .

Sur le même mode, il existe le mot clé *or* qui signifie cette fois "ou". Nous allons prendre le même exemple, sauf que nous allons monter notre condition différemment.

Nous allons chercher à savoir si *a* n'est pas dans l'intervalle. La variable ne se trouve pas dans l'intervalle si elle est inférieur à 2 ou supérieur à 8. Voici donc le code :

Code : Python

```
if a<2 or a>8:
    print("a n'est pas dans l'intervalle.")
else:
    print("a est dans l'intervalle.")
```

Enfin, il existe le mot clé *not* qui "inverse" un prédicat. Le prédicat `not a==5` équivaut donc à `a!=5`.

not rend la syntaxe plus claire. Pour cet exemple, j'ajoute à cette liste un nouveau mot clé *is* qui teste non pas l'égalité des valeurs de deux variables, mais l'égalité de leur référence. Je ne vais pas rentrer dans le détail de ce mécanisme avant longtemps. Qu'il vous suffise de savoir que pour les entiers, les flottants et les booléens, c'est strictement la même chose. Mais en testant une égalité entre variables dont le type est plus complexe, préférez l'opérateur "==". Mais revenons à cette démonstration :

Code : Python Console

```
>>> majeur = False
>>> if majeur is not True:
...     print("Vous n'êtes pas encore majeur.")
...
Vous n'êtes pas encore majeur.
>>>
```

En parlant un minimum l'anglais, ce prédicat est limpide, et d'une simplicité sans égale .

Vous pouvez tester les prédicats plus complexes de la même façon que les précédents, en les entrant directement, sans le if ni les deux points dans l'interpréteur de commande. Vous pouvez utiliser les parenthèses ouvrantes et fermantes pour encadrer des prédicats et les comparer selon une priorité bien précise (nous verrons ce point plus loin si vous n'en comprenez pas l'utilité).

Votre premier programme !

À quoi on joue ?

L'heure est venue pour le premier TP. S'agissant du tout premier, et parce qu'il y a quelques indications que je dois vous donner pour que vous le réalisiez jusqu'au bout, je vous accompagnerai pas à pas dans sa réalisation.

Avant de commencer

Vous allez dans cette sous-partie écrire votre premier programme. Vous allez sûrement tester les syntaxes dans l'interpréteur de commande directement. Mais vous pourriez préférer écrire votre code directement dans un fichier que vous pourrez exécuter. Si c'est le cas, je vous renvoie au récapitulatif traitant de ce point.

Sujet

Le but de notre programme va être de déterminer si une année entrée par l'utilisateur est bissextile. Il s'agit d'un sujet très prisé des enseignants en informatique quand il s'agit d'expliquer les conditions. Mille pardons donc à ceux qui ont déjà fait cet exercice dans un autre langage, mais, une fois n'est pas coutume, je trouve que ce petit programme reprend assez de thèmes abordés dans ce chapitre pour être réellement intéressant.

Je vous rappelle les règles qui déterminent si une année est bissextile ou non (vous allez peut-être même apprendre des choses que le commun des mortels ignore).

Une année est dite bissextile si c'est un multiple de 4, sauf si c'est un multiple de 100. Toutefois, elle est considérée comme bissextile si c'est un multiple de 400. Je développe :

- Si une année n'est pas multiple de 4, on s'arrête là, elle n'est pas bissextile.
- Si elle l'est, on regarde si elle est multiple de 100.
    - Si c'est le cas, on regarde si elle est multiple de 400.

        - Si c'est le cas, l'année est bissextile.
        - Sinon, elle ne l'est pas.

    - Sinon, elle l'est.

Solution ou résolution

Voilà. Le problème est posé clairement (sinon relisez attentivement l'énoncé autant de fois que nécessaire), il faut maintenant réfléchir à sa résolution en terme de programmation. C'est une phase de transition assez délicate de prime abord, et je vous conseille de schématiser le problème, de prendre des notes sur les différentes étapes, sans pour l'instant penser au code. C'est une phase purement algorithmique, autrement dit, on réfléchit au programme sans réfléchir au code.

Vous aurez besoin, pour réaliser ce petit programme, de quelques indications qui sont réellement spécifiques à Python. Ne lisez donc ceci qu'après avoir cerné et clairement écrit le problème d'une façon plus algorithmique. Cela étant dit, si vous peinez à trouver une solution, ne vous y attardez pas. Cette phase de réflexion est assez difficile au début, et parfois il suffit d'un peu de pratique et d'explications pour comprendre l'essentiel.

La fonction input()

Tout d'abord, j'ai fais mention d'une année entrée par l'utilisateur. En effet, depuis tout à l'heure, on teste des variables que l'on déclare nous-mêmes avec une valeur précise. La condition est donc assez ridicule.

input() est une fonction qui va, pour nous, caractériser nos premières interactions avec l'utilisateur : le programme réagira différemment en fonction du nombre que l'utilisateur entrera.

input() ne prend aucun paramètre. Elle interrompt le programme et attend que l'utilisateur entre ce qu'il veut puis appuie sur Entrée. A cet instant, la fonction retourne ce que l'utilisateur a entré. Il faut donc piéger cette valeur dans une variable.

Code : Python Console

```
>>> # test de la fonction input
>>> print("Entrez une année :")
>>> annee = input()
2009
>>> print(annee)
'2009'
>>>
```

Il subsiste un problème : le type de la variable annee après l'appel à input() est... une chaîne de caractère  . Vous pouvez vous en rendre compte grâce aux apostrophes qui encadrent la valeur de la variable quand vous l'affichez directement dans l'interpréteur.

C'est bien ennuyeux tout ça, nous qui allons devoir travailler sur un entier, nous allons devoir convertir cette variable. Pour convertir une variable dans un autre type, il faut utiliser le nom du type comme une fonction (ce que c'est, d'ailleurs).

Code : Python Console

```
>>> type(annee)
<type 'str'>
>>> # on veut convertir la variable en un entier, on utilise donc
>>> # la fonction int qui prend en paramètre la variable d'origine
>>> annee = int(annee)
>>> type(annee)
<type 'int'>
>>> print(annee)
2009
>>>
```

Bon, parfait ! On a donc maintenant l'année sous sa forme entière. Notez que si vous entrez des lettres lors de l'appel à *input()*, la conversion renverra une erreur.

L'appel à la fonction *int()* en a peut-être déconcerté certains. On passe en paramètre de cette fonction la variable contenant la chaîne de caractère issue de *input()*, pour tenter de la convertir. La fonction *int()* retourne la valeur convertie en entier, et on la récupère donc dans la même variable. On évite ainsi de travailler sur plusieurs variables, sachant que la première n'a plus aucune utilité à présent qu'on l'a convertie.

Test de multiples

Certains pourraient également se demander comment tester si un nombre a est multiple d'un nombre b. Il suffit en fait de tester le reste de la division entière de b par a. Si ce reste est nul, alors a est un multiple de b.

Code : Python Console

```
>>> 5%2 # 5 n'est pas un multiple de 2
1
>>> 8%2 # 8 est un multiple de 2
0
>>>
```

A vous de jouer

Je pense vous avoir donné le nécessaire pour réussir. A mon avis, le plus difficile est la phase de réflexion qui précède la composition du programme. Si vous avez du mal à réaliser cette opération, passez à la correction et étudiez-la soigneusement. Sinon, on se retrouve à la partie suivante.

Bonne chance ! 

Correction

C'est l'heure de comparer nos méthodes, et avant de vous divulguer le code de ma solution, je vous précise qu'elle est loin d'être la seule possible et que vous pouvez très bien avoir trouvé quelque chose de différent, mais qui marche tout aussi bien.

Attention... la voiiiciiiiiii...

Code : Python

```
# programme testant si une année, entrée par l'utilisateur,
# est bissextile ou non

print("Entrez une année :")
annee = input() # on attend que l'utilisateur entre l'année qu'il
désire tester
annee = int(annee) # risque d'erreur si l'utilisateur n'a pas rentré
un nombre
bissextile = False # on crée un booléen qui va être vrai ou faux
                   # si l'année est bissextile ou non

if annee%400==0:
    bissextile = True
elif annee%100==0:
    bissextile = False
elif annee%4==0:
    bissextile = True
else:
    bissextile = False
if bissextile: # si l'année est bissextile
    print("L'année entrée est bissextile.")
else:
```

```
        print("L'année entrée n'est pas bissextile.")
```

Je pense que le code est assez clair, reste à expliciter l'enchaînement des conditions. Vous remarquerez qu'on a inversé le problème. On test en effet d'abord si l'année est un multiple de 400, ensuite si c'est un multiple de 100, et ensuite si c'est un multiple de 4. En effet, le "elif" garantit que, si annee est un multiple de 100, ce n'est pas un multiple de 400 (car le cas a été traité au-dessus). De cette façon, on s'assure que tous les cas sont prévus. Vous pouvez faire des essais avec plusieurs années et vous rendre compte si le programme a raison ou pas.

N.B.: l'utilisation de *bissextile* comme d'un prédicat à part entière vous a peut-être déconcerté. C'est en fait tout à fait possible et logique, puisque *bissextile* est un booléen. Il est de ce fait vrai ou faux et donc on peut le tester simplement. On peut bien entendu aussi écrire `if bissextile==True:`, cela revient au même.

Un peu d'optimisation

Ce qu'on a fait était bien, mais on peut l'améliorer. D'ailleurs, vous vous rendrez compte que c'est presque toujours le cas. Ici, il s'agit bien entendu de notre condition que je vais passer au crible afin d'en construire si possible une plus courte et plus logique. On peut parler d'optimisation dans ce cas, même si l'optimisation intègre aussi et surtout les ressources consommées par votre application, en vue de les diminuer et d'améliorer sa rapidité. Mais pour une petite application comme cela, je ne pense pas qu'on perdra du temps sur l'optimisation du temps d'exécution  .

Le premier détail que vous auriez pu remarquer, c'est que le "else" de fin est inutile. En effet, la variable *bissextile* est à *False* par défaut et le reste donc si le cas n'est pas traité (ici, quand l'année n'est ni un multiple de 400, ni un multiple de 100, ni un multiple de 4).

Ensuite, il apparaît que nous pouvons faire un grand ménage dans notre condition, car, les deux seuls cas dans lesquels l'année est bissextile sont "si l'année est un multiple de 400" ou "si l'année est un multiple de 4 mais pas de 100".

Le prédicat correspondant est un peu délicat, il fait appel aux priorités des parenthèses. Je n'attendais pas que vous le trouviez tout seul, mais que vous le compreniez bien à présent.

Code : Python

```
# programme testant si une année, entrée par l'utilisateur,
# est bissextile ou non

print("Entrez une année :")
annee = input() # on attend que l'utilisateur entre l'année qu'il
désire tester
annee = int(annee) # risque d'erreur si l'utilisateur n'a pas rentré
un nombre
                   # si l'année est bissextile ou non

if annee%400==0 or (annee%4==0 and annee%100!=0):
    print("L'année entrée est bissextile.")
else:
    print("L'année entrée n'est pas bissextile.")
```

Du coup, on n'a plus besoin de la variable *bissextile*, c'est déjà ça de gagné. Nous sommes passés de 16 lignes de code à seulement 7 (sans compter les commentaires et les sauts de ligne) ce qui n'est pas rien  .

Et voilà, ça fait pas mal de choses à ingérer déjà. Dès que vous vous sentez prêt, passez au chapitre suivant, nous attaquons les boucles, un autre grand concept de la programmation. Rassurez-vous, on se dirige lentement mais sûrement vers vos premiers véritables programmes :).

Les boucles

Les boucles sont un nouveau concept qui va vous permettre de répéter une certaine opération autant de fois que nécessaire. Le concept risque de vous sembler un peu théorique, car les applications pratiques qui seront présentées dans ce chapitre ne vous paraitront probablement pas très intéressantes. Toutefois, il est impératif que cette notion soit comprise avant que vous ne passiez à la suite. Viendra vite le moment où vous aurez du mal à écrire une application sans boucle. Alors, on commence ?

En quoi ça consiste ?

Comme je l'ai dit juste au-dessus, les boucles sont un moyen de répéter des instructions de votre programme un certain nombre de fois. Prenons un exemple simple, même si assez peu réjouissant en lui-même : écrire un programme affichant la table de multiplication par 7, de 1 * 7 à 10 * 7.

... bah quoi ?

Bon, ce n'est qu'un exemple, ne faites pas cette tête, et puis je suis sûr que ce sera utile pour certains . Dans un premier temps, vous devriez arriver au programme suivant :

Code : Python

```
print(" 1 * 7 =",1*7)
print(" 2 * 7 =",2*7)
print(" 3 * 7 =",3*7)
print(" 4 * 7 =",4*7)
print(" 5 * 7 =",5*7)
print(" 6 * 7 =",6*7)
print(" 7 * 7 =",7*7)
print(" 8 * 7 =",8*7)
print(" 9 * 7 =",9*7)
print("10 * 7 =",10*7)
```

... et le résultat :

Code : Console

```
1 * 7 = 7
 2 * 7 = 14
 3 * 7 = 21
 4 * 7 = 28
 5 * 7 = 35
 6 * 7 = 42
 7 * 7 = 49
 8 * 7 = 56
 9 * 7 = 63
10 * 7 = 70
```

Bon, c'est sûrement la première idée qui vous est venue, et ça fonctionne, très bien même. Seulement, vous reconnaitrez qu'un programme comme ça n'est pas bien utile. Si on compte partir d'une variable à présent, une variable qui contiendrait le 7 (comme ça, si on décide d'afficher la table de multiplication de 6, on n'aura qu'à changer la valeur de la variable). Pour cet exemple, on utilise une variable *nb* qui contiendra 7. Les instructions seront légèrement différentes, mais vous devriez toujours pouvoir écrire ce programme :

Code : Python

```
nb = 7
print(" 1 *", nb, "=", 1*nb)
print(" 2 *", nb, "=", 2*nb)
print(" 3 *", nb, "=", 3*nb)
print(" 4 *", nb, "=", 4*nb)
print(" 5 *", nb, "=", 5*nb)
print(" 6 *", nb, "=", 6*nb)
```

```
print(" 7 *", nb, "=", 7*nb)
print(" 8 *", nb, "=", 8*nb)
print(" 9 *", nb, "=", 9*nb)
print("10 *", nb, "=",10*nb)
```

Le résultat est le même, vous pouvez vérifier. Mais le code est quand-même un peu plus intéressant : on peut changer la table de multiplication à afficher en changeant la variable *nb*.

Mais ce programme reste assez peu pratique. Il fait une action bien répétitive, et les programmeurs sont très paresseux, d'où les boucles.

La boucle while

La boucle que je vais présenter se retrouve dans la plupart des autres langages de programmation, et sous le même nom. Elle permet de répéter un **bloc d'instructions** tant qu'une condition est vraie (*while* signifie "tant que" en anglais). J'espère que le concept de **bloc d'instructions** est clair pour vous, sinon, je vous renvoie au chapitre précédent.

La syntaxe du *while* est :

Code : Python

```
while condition:
    # instruction 1
    # instruction 2
    # ...
    # instruction N
```

Vous devriez reconnaitre la forme d'un bloc d'instructions, du moins je l'espère .

Quelle condition va-t-on utiliser ?

Et bien, c'est là le point important. On va dans cet exemple créer une variable que l'on va incrémenter dans le bloc d'instructions. Tant que cette variable sera inférieure à 10, le bloc s'exécutera pour afficher la table.

Si ce n'est pas clair, regardez ce code, quelques commentaires suffiront pour le comprendre :

Code : Python

```
nb = 7 # on garde la variable contenant le nombre dont on veut la
table de multiplication
i = 0 # c'est notre variable compteur que nous allons incrémenter
dans la boucle

while i<10: # tant que i est strictement inférieure à 10
    print(i+1 , "*" , nb , "=" , (i+1)*nb)
    i += 1 # on incrémente i de 1 à chaque tour de boucle
```

Prenons ce code ligne par ligne :

1. On instancie notre variable *nb* qui va accueillir le nombre sur lequel nous allons travailler (7 en l'occurence). Vous avez bien entendu la possibilité de faire entrer ce nombre à l'utilisateur, vous savez le faire à présent.
2. On instancie la variable *i* qui sera notre compteur durant la boucle. *i* est un standard utilisé quand il s'agit de boucles et de variables s'incrémentant, mais il va de soi que vous auriez pu lui donner un autre nom. On l'initialise à 0.
3. Un saut de ligne ne fait jamais de mal !
4. On trouve ici l'instruction *while* qui se décode, comme je l'ai indiqué en commentaire, en **"tant que *i* est strictement**

inférieure à 10". N'oubliez pas les deux points à la fin de la ligne.
5. La ligne du *print*, vous devez la reconnaitre. Maintenant, la plus grande partie de la ligne affichée est constituée de variables, à part les signes mathématiques. Vous remarquez qu'à chaque fois qu'on utilise *i* dans cette ligne, pour l'affichage ou le calcul, on lui ajoute 1. Cela est dû au fait qu'en programmation, on a l'habitude, (habitude que vous devrez prendre) de commencer à compter à partir de 0. Seulement ce n'est pas le cas de la table de multiplication, qui va de 1 à 10 et non de 0 à 9, comme c'est le cas ici. Certes j'aurais pu changer la condition et la valeur initiale de *i*, ou même mettre l'incrémentation de *i* avant l'affichage, mais j'ai voulu prendre le cas le plus utilisé, le format de boucle que vous retrouverez le plus souvent. Rien ne vous empêche de faire les tests, et je vous y encourage même.
6. Ici, on incrémente la variable *i* de 1. Si on est dans le premier tour de boucle, *i* passe donc de 0 à 1. Et alors, puisqu'il s'agit de la fin du bloc d'instructions, on revient à l'instruction *while*. *while* vérifie que *i* est toujours inférieure à 10. Si c'est le cas (et ça l'est pour l'instant), on réexécute le bloc d'instructions. En tout, on exécute ce bloc 10 fois, jusqu'à ce que *i* passe de 9 à 10. Alors, l'instruction *while* vérifie la condition, se rend compte qu'elle est à présent fausse (*i* n'est pas inférieure à 10, puisqu'elle est maintenant égale à 10) et s'arrête. S'il y avait du code après le bloc, il serait à présent exécuté.

N'oubliez pas d'incrémenter *i* ! Sinon, vous créerez ce qu'on appelle une boucle infinie, puisque *i* ne sera jamais supérieure à 10, et que la condition du *while* sera toujours vraie... Elle s'exécutera donc à l'infini, du moins hypothétiquement. Si votre ordinateur se lance dans une boucle infinie à cause de votre programme, vous devrez taper, sous Windows ou Linux, ctrl + C dans la fenêtre de l'interpréteur pour interrompre la boucle. Python ne le fera pas tout seul, car pour lui, il se passe bel et bien quelque chose, et de toute façon il est incapable de différencier une boucle infinie d'une boucle finie : c'est au programmeur de le faire.

La boucle for

L'instruction *while*, ai-je dit, est la plus présente dans la plupart des autres langages. Dans le C++ ou le Java, on retrouve également des instructions *for* mais qui n'ont pas le même sens. C'est assez particulier et c'est le point sur lequel je risque de manquer d'exemples pour l'illustrer, toute son utilité se révélera au chapitre sur les listes. Notez que, si vous avez fait du Perl ou du PHP, vous pouvez retrouver les boucles *for* sous un mot-clé assez proche : *foreach*.

L'instruction *for* travaille sur des ensembles de données. Elle est en fait spécialisée dans le parcours d'un ensemble de données. Nous n'avons pas vu (et nous ne verrons pas tout de suite), ces ensembles assez particuliers et utilisés, mais aussi complexes. Toutefois, il y en a un que nous avons vu depuis quelque temps déjà : les chaînes de caractères.

Les chaînes de caractères sont des ensembles... de caractères  ! Vous pouvez parcourir une chaîne de caractères, et cela était rendu possible également par *while*, nous verrons plus tard comment. Pour l'instant, intéressons-nous à `for` .

L'instruction *for* se construit ainsi :

Code : Python

```
for element in ensemble:
```

element est une variable créée par le *for*, ce n'est pas à vous de l'instancier. Elle prend successivement les valeurs contenues dans l'ensemble parcouru.

Ce n'est pas très clair ? Alors, comme d'habitude, tout s'éclaire avec le code ! 

Code : Python

```
chaine = "Bonjour les ZER0S"
for lettre in chaine:
    print(lettre)
```

Code : Console

```
B
o
n
```

```
j
o
u
r

l
e
s

Z
E
R
0
S
```

Est-ce plus clair ? En fait, la variable *lettre* prend successivement la valeur de chaque lettre contenue dans la chaîne de caractères (d'abord B, puis o, puis n...). On les affiche avec `print` et cette fonction fait un retour à la ligne après chaque message, ce qui fait que toutes les lettres sont sur une seule colonne. Littéralement, la ligne 2 signifie **"pour lettre dans chaine"**. Arrivé à cette ligne, l'interpréteur va créer une variable *lettre* qui contiendra le premier élément de la chaîne (la première lettre, autrement dit). Après l'exécution du bloc, la variable "lettre" contient la seconde lettre, et ainsi de suite, tant qu'il y a une lettre dans la chaîne.

Notez bien que du coup, il est inutile d'incrémenter la variable *lettre* (ce qui serait d'ailleurs assez ridicule vu que ce n'est pas un nombre). Python se charge de l'incrémentation, un des grands avantages de l'instruction *for*.

A l'instar des conditions que nous avons vu jusqu'ici, `in` peut être utilisée ailleurs que dans une boucle *for*.

Code : Python

```
chaine = "Bonjour les ZER0S"
for lettre in chaine:
    if lettre in "AEIOUYaeiouy": # lettre est une voyelle
        print(lettre)
    else: # lettre est une consonne... ou plutôt, lettre n'est pas
une voyelle
        print("*")
```

... ce qui donne :

Code : Console

```
*
o
*
*
o
u
*
*
*
e
*
*
*
E
*
*
*
```

Voilà ! Soit l'interpréteur affiche les lettres si ce sont des voyelles, soit il affiche des "*". Notez bien que le 0 n'est pas affiché à la fin, Python ne se doute nullement qu'il s'agit d'un "o" stylisé .

Retenez bien cette utilisation de *in* dans une condition. On cherche à savoir si un élément quelconque est contenu dans un ensemble quelconque (ici si la lettre est contenue dans "AEIOUYaeiouy", c'est-à-dire si lettre est une voyelle). On retrouvera plus loin cette fonctionnalité.

Un petit bonus : les mot-clés break et continue

Je vais ici vous montrer deux nouveaux mot-clés, *break* et *continue*. Vous ne les utiliserez peut-être pas beaucoup, mais vous devez au moins savoir qu'ils existent... et à quoi ils servent .

break

Le mot-clé *break* permet tout simplement d'interrompre une boucle. Il est souvent utilisé dans une forme de boucle que je n'approuve pas trop :

Code : Python

```
while 1: # 1 est toujours vrai -> boucle infinie
    print("Entrez 'Q' pour quitter")
    lettre = input()
    if lettre=="Q":
        print("Fin de la boucle")
        break
```

La boucle *while* a pour condition *1*, c'est-à-dire une condition qui sera toujours vraie. Autrement dit, en regardant la ligne du *while*, on pense à une boucle infinie. En pratique, on demande à l'utilisateur d'entrer une lettre (un 'Q' pour quitter). Tant que l'utilisateur n'entre pas cette lettre, le programme lui redemande d'entrer une lettre. Quand il entre 'Q', le programme affiche `Fin de la boucle` et la boucle s'arrête grâce au mot-clé *break*.

Ce mot-clé permet d'arrêter une boucle quel que soit la condition de la boucle. Python sort immédiatement de la boucle et exécute le code qui se trouve ensuite, si il y en a.

C'est un exemple un peu simpliste, mais vous pouvez voir l'idée d'ensemble. Dans ce cas-là et, à mon sens, dans la plupart des cas ou *break* est utilisé, on pourrait s'en sortir en précisant une véritable condition à la ligne du *while*. Par exemple, pourquoi ne pas créer un booléen qui sera *vrai* tout au long de la boucle et *faux* quand la boucle doit s'arrêter ? Ou bien tester directement si *lettre != "Q"* dans le *while* ?

Parfois, *break* est véritablement utile et fait gagner du temps. Mais ne l'utilisez pas à outrance, préférez une boucle avec une condition claire plutôt qu'un bloc d'instructions avec un *break*, qui sera plus dure à appréhender d'un seul regard.

continue

Le mot-clé *continue* permet de... continuer une boucle, en repartant directement à la ligne du *while* ou *for*. Un petit exemple s'impose je pense :

Code : Python

```
i=1
while i<20: # tant que i est inférieure à 20
    if i%3==0:
        i+=4 # on ajoute 4 à i
        print("On incrémente i de 4. i est maintenant égale à", i)
        continue # on retourne au while sans exécuter les autres
lignes
    print("La variable i =", i)
    i+=1 # dans le cas classique on ajoute juste 1 à i
```

Voici le résultat :

Code : Console

```
La variable i = 1
La variable i = 2
On incrémente i de 4. i est maintenant égale à 7
La variable i = 7
La variable i = 8
On incrémente i de 4. i est maintenant égale à 13
La variable i = 13
La variable i = 14
On incrémente i de 4. i est maintenant égale à 19
La variable i = 19
```

Comme vous le voyez, tous les trois tours de boucle, *i* s'incrémente de 4. Arrivé au mot-clé *continue*, Python n'exécute pas la fin du bloc mais revient au début de la boucle en retestant la condition du *while*. Autrement dit, quand Python arrive à la ligne 6, il saute à la ligne 2, sans exécuter la ligne 7 et 8. Au nouveau tour de boucle, Python reprend l'exécution normale de la boucle (*continue* n'ignore la fin du bloc que pour le tour de boucle courant).

Mon exemple ne démontre pas de manière éclatante l'utilité de *continue*. Les rares fois où j'utilise ce mot-clé, c'est par exemple pour supprimer des éléments d'une liste, mais nous n'avons pas encore vu les listes  . L'essentiel pour l'instant c'est que vous vous souveniez de ces deux mot-clés et que vous sachiez ce qu'ils font, si vous les rencontrez au détour d'une instruction  .
Personnellement, je n'utilise pas très souvent ces mot-clés, mais c'est aussi une question de goût.
Nous avons vu ce que je considère comme la base des boucles dans ce chapitre. Dans les parties suivantes, nous reviendrons à ce concept pour l'agrémenter de nouveaux exemples et nouvelles fonctionnalités. Retenez bien les deux formes de boucles, *while* et *for*, sachant qu'à l'avenir nous travaillerons surtout avec *for*. Retenez également le mot-clé *in* et son utilité, nous le reverrons dans la seconde partie, notamment dans le chapitre consacré aux listes.

Pas à pas vers la modularité (1/2)

En programmation, on est souvent amené à utiliser plusieurs fois des groupes d'instructions pour un but très précis. Ce n'est pas de boucles dont je parle ici. Simplement, vous pourrez vous rendre compte que la plupart de nos tests pourront être regroupés dans des blocs plus vastes, fonctions, ou modules. Je vais détailler tranquillement ces deux concepts.

Vous découvrirez beaucoup de choses nouvelles dans ce chapitre, alors prenez bien le temps de le lire et le comprendre, sans précipitation.

Les fonctions : à vous de jouer

Nous avons utilisé pas mal de fonctions depuis le début de ce tutoriel. On citera pour mémoire **print**, type et input, sans compter quelques autres. Mais vous devez bien vous rendre compte qu'il existe un nombre incalculable de fonctions déjà construites en Python. Toutefois, vous vous apercevrez aussi que très souvent on crée nos propres fonctions. C'est le premier pas que vous ferez, dans ce chapitre, vers la **modularité**. Ce terme un peu barbare signifie que l'on va s'habituer à regrouper des parties de notre code que nous serons amenés à réutiliser dans des fonctions. Dans le chapitre suivant, nous apprendrons à regrouper nos fonctions ayant un rapport dans un fichier pour constituer un module, mais n'anticipons pas.

La création de fonctions

Nous allons pour illustrer cet exemple reprendre le code de la table de multiplication, que nous avons vu dans le chapitre précédent et qui, décidément, n'en finit pas de vous poursuivre  .

Nous allons emprisonner notre code calculant la table de multiplication par 7 dans une fonction que nous appellerons *table_par_7*.

On crée une fonction selon le schéma suivant :

Code : Python

```
def nom_de_la_fonction(parametre1, parametre2, parametre3,
parametreN):
    # bloc d'instructions
```

Les blocs d'instructions nous courent après aussi, quel enfer  . Si l'on décortique la ligne de la définition de la fonction, on trouve dans l'ordre :

- **def** : le mot-clé qui est l'abréviation de "define" (définir en anglais) et qui est le prélude à toute construction de fonction.
- Le nom de la fonction, qui se nomme exactement comme une variable (nous verrons par la suite que ce n'est pas par hasard). N'utilisez pas un nom de variable déjà instanciée pour nommer une fonction.
- La liste des paramètres qui seront fournis à la fonction lors de son appel, séparés par une virgule et encadrée par une parenthèse ouvrante et une fermante (là encore, les espaces sont optionnelles mais aident à la lisibilité).
- Les deux points, encore et toujours, qui clôturent la ligne.

N.B. : les parenthèses sont obligatoires, quand bien même votre fonction n'attendrait aucun paramètre.

Le code pour mettre notre table de multiplication par 7 dans une fonction serait donc :

Code : Python

```
def table_par_7():
    nb = 7
    i = 0 # Notre compteur ! L'auriez-vous oublié ?
    while i<10: # Tant que i est strictement inférieure à 10,
        print(i+1 , "*" , nb , "=" , (i+1)*nb)
        i += 1 # on incrémente i de 1 à chaque tour de boucle.
```

Quand vous exécutez ce code à l'écran, rien ne se passe. Une fois que vous avez retrouvé les trois chevrons essayez d'appeler la fonction :

Code : Python Console

```
>>> table_par_7()
1 * 7 = 7
2 * 7 = 14
3 * 7 = 21
4 * 7 = 28
5 * 7 = 35
6 * 7 = 42
7 * 7 = 49
8 * 7 = 56
9 * 7 = 63
10 * 7 = 70
>>>
```

Bien, c'est, euh, exactement ce qu'on avait réussi à faire dans le chapitre précédent, et l'intérêt ne saute pas encore aux yeux, sauf que l'on peut appeler facilement la fonction et réafficher toute la table sans avoir besoin de tout réécrire. Enfin à part ça...

Mais, si on entre des paramètres pour pouvoir afficher la table de 5 ou de 8... ?

Oui, ce serait déjà bien plus utile. Je ne pense pas que vous ayez trop de mal à trouver le code de la fonction :

Code : Python

```
def table(nb):
    i = 0
    while i<10: # Tant que i est strictement inférieure à 10,
        print(i+1 , "*" , nb , "=" , (i+1)*nb)
        i += 1 # on incrémente i de 1 à chaque tour de boucle.
```

Et là, vous pouvez entrer différents nombres en paramètres, `table(8)` pour afficher la table de multiplication par 8 par exemple.

On peut aussi envisager de passer en paramètre le nombre de valeurs à afficher dans la table.

Code : Python

```
def table(nb, max):
    i = 0
    while i<max: # Tant que i est strictement inférieure à la
variable max,
        print(i+1 , "*" , nb , "=" , (i+1)*nb)
        i += 1
```

Si vous entrez à présent `table(11, 20)`, l'interpréteur vous affichera la table de 11, de 1*11 à 20*11. Magique non ? 

Dans le cas où on utilise plusieurs paramètres sans les nommer, comme ici, il faut respecter l'ordre d'appel des paramètres, cela va de soi. Si vous commencez à mettre le nombre d'affichages en premier paramètre alors que dans la définition c'était le second, vous risquez d'avoir quelques surprises  . Il est possible d'appeler les paramètres dans le désordre mais il faut dans ce cas préciser leur nom, nous verrons ça plus bas.

Si vous précisez en second paramètre un nombre négatif, vous avez toutes les chances de créer une magnifique boucle infinie... vous pouvez l'empêcher en rajoutant des vérifications avant la boucle, comme par exemple si le nombre est négatif ou nul, je le mets à 10. D'un autre côté, ça va de soi qu'on ne précise pas de nombre négatif quand on vous demande un maximum dans ce cadre. En Python, on préférera mettre un commentaire en tête de fonction ou une **docString** comme on le verra plus bas, pour indiquer que max doit être positif, plutôt que de faire des vérifications qui au final feront perdre du temps. Une des phrases reflettant la philosophie du langage et qui peut s'appliquer à ce type de situation est *we're all consenting adults here.* : nous sommes entre adultes consentants (sous-entendu, quelques avertissements en commentaires sont plus efficaces qu'une restriction au niveau du code). On aura l'occasion de retrouver cette phrase plus loin, surtout quand on parlera des objets.

Valeurs par défaut des paramètres

On peut également préciser une valeur par défaut aux paramètres de la fonction. Vous pouvez par exemple dire que le nombre maximum d'affichages doit être de 10 par défaut (c'est-à-dire si l'utilisateur de votre fonction ne le précise pas). Cela se fait le plus simplement du monde :

Code : Python

```
def table(nb, max=10):
    """Fonction affichant la table de multiplication par nb
de 1*nb à max*nb

(max >= 0)
"""
    i = 0
    while i<max:
        print(i+1 , "*" , nb , "=" , (i+1)*nb)
        i += 1
```

Il suffit de rajouter *=10* après max. A présent, vous pouvez appeler la fonction de deux façons : soit en précisant le numéro de la table et le nombre maximum d'affichages, soit en ne précisant que le numéro de la table (`table(7)`). Dans ce dernier cas, max vaudra 10 par défaut.

J'en ai profité pour ajouter quelques lignes d'explications que vous aurez remarquées. Nous avons mis une chaîne de caractères, sans la capturer dans une variable, juste en-dessous de la définition de la fonction. Cette chaîne est ce qu'on appelle une **docstring** que l'on pourrait traduire par une chaîne d'aide. Si vous faites help(table), c'est ce message que vous verrez apparaître. Documenter vos fonctions est également une bonne habitude à prendre. Comme vous le voyez on indente cette chaîne et on la met entre triple guillemets. Si la chaîne est sur une seule ligne on pourra mettre les trois guillemets clôturant la chaîne sur la même ligne, sinon on préférera sauter une ligne avant de fermer cette chaîne pour des raisons de lisibilité. Tout le texte d'aide est indenté au même niveau que le code de la fonction.

Enfin, sachez que l'on peut appeler des paramètres par leur nom. Cela est utile pour une fonction comptant un certain nombre de paramètres qui ont tous une valeur par défaut. Vous pouvez aussi utiliser cette méthode sur une fonction sans paramètre par défaut, mais c'est moins courant.

Prenons un exemple de définition de fonction :

Code : Python

```
def fonc(a=1, b=2, c=3, d=4, e=5):
    print("a =",a,"b =",b,"c =",c,"d =",d,"e =",e)
```

Simple, n'est-ce pas ? Et bien, vous avez de nombreuses façons d'appeler cette fonction. En voici quelques exemples :

Instruction	Résultat
`fonc()`	a = 1 b = 2 c = 3 d = 4 e = 5
`fonc(4)`	a = 4 b = 2 c = 3 d = 4 e = 5
`fonc(b=8, d=5)`	a = 1 b = 8 c = 3 d = 5 e = 5

`fonc(b=35, c=48, a=4, e=9)`	a = 4 b = 35 c = 48 d = 4 e = 9

Je ne pense pas que des explications suplémentaires s'imposent. Si vous voulez changer la valeur d'un paramètre, vous entrez son nom, un signe égal, et une valeur (qui peut être une variable bien entendu). Peu importe les paramètres que vous précisez (comme vous le voyez, dans cet exemple ou tous les paramètres ont une valeur par défaut, vous pouvez appeler la fonction sans paramètres), peu importe l'ordre d'appel des paramètres.

Signature d'une fonction

On entend par "signature de fonction" les éléments qui permettent au langage de l'identifier. En C++ par exemple, la signature d'une fonction est constituée de son nom et des types de ses paramètres. Cela veut dire que l'on peut trouver plusieurs fonctions du même nom mais avec différents paramètres dans leur définition. Au moment de l'appel de fonction, le compilateur recherche la fonction qui s'applique à cette signature.

En Python comme vous avez pu le voir, on ne précise pas les types des paramètres. Dans ce langage, la signature d'une fonction est tout simplement son nom. Cela signifie que vous ne pouvez définir deux fonctions du même nom (l'ancienne définition est écrasée par la nouvelle si vous le faites).

Code : Python

```
def exemple():
    print("Un exemple d'une fonction sans paramètres")

exemple()

def exemple(): # on redéfinit la fonction exemple
    print("Un autre exemple de fonction sans paramètres")

exemple()
```

A la ligne 1 on définit la fonction *exemple*. On l'appelle une première fois à la ligne 4. On redéfinit à la ligne 6 la fonction *exemple*. L'ancienne définition est écrasée et l'ancienne fonction ne pourra plus être appelée.

Retenez simplement que, comme pour les variables, un nom de fonction ne renvoie que vers une fonction précise, on ne peut surcharger de fonctions en Python.

L'instruction `return`

Ce que nous avons fait était intéressant, mais nous n'avons pas encore fait le tour des possibilités de la fonction. Et d'ailleurs, même à la fin de ce chapitre, il nous restera quelques petites fonctionnalités à voir. Si vous vous souvenez bien, il existe des fonctions comme `print` qui ne retournent rien (attention, retourner et afficher ne sont pas identiques) et des fonctions, telles que `input` ou `type` qui retournent une valeur. Vous pouvez capturer cette valeur en mettant une variable devant (exemple `variable2 = type(variable1)`). En effet, les fonctions travaillent en général sur des données et retournent le résultat obtenu, suite à un calcul par exemple.

Prenons un exemple simple : une fonction chargée de mettre au carré une valeur passée en paramètre. Je vous rappelle que Python en est parfaitement capable sans avoir à coder une nouvelle fonction, mais c'est pour l'exemple  .

Code : Python

```
def carre(valeur):
    return valeur * valeur
```

L'instruction `return` signifie qu'on va **retourner** la valeur, pour pouvoir la récupérer ensuite et la stocker dans une variable par exemple. Cette instruction arrête le déroulement de la fonction, du code placé après le return ne s'exécutera pas.

Code : Python

```
variable = carre(5)
```

La variable *variable* contiendra après cette instruction 5 au carré, c'est-à-dire 25.

Sachez que l'on peut retourner plusieurs valeurs que l'on sépare par des virgules, et que l'on peut les capturer dans des variables également séparées par des virgules, mais je m'attarderai plus loin sur cette particularité. Retenez simplement la définition d'une fonction, les paramètres, les valeurs par défaut, l'instruction **`return`** et ce sera déjà bien  .

Les fonctions lambda

Nous venons de voir comment créer une fonction grâce au mot-clé *def*. Python nous propose un autre moyen de créer des fonctions, des fonctions extrêmement courtes car limitées à une seule instruction.

 Pourquoi une autre façon de créer des fonctions ? La première suffit non ?

Disons que ce n'est pas tout à fait la même chose, comme vous allez le voir. Les fonctions lambda sont en général utilisées dans un certain contexte pour lequel définir une fonction grâce à *def* serait plus long et moins pratique.

Syntaxe

Avant tout, voyons la syntaxe d'une définition de fonction lambda. Nous allons utiliser le mot-clé *lambda* comme ceci :

Citation

lambda arg1, arg2, ...: instruction de retour

Je pense qu'un exemple vous semblera plus clair. On veut créer une fonction qui prend un paramètre et retourne ce paramètre au carré.

Code : Python Console

```
>>> lambda x: x * x
<function <lambda> at 0x00BA1B70>
>>>
```

D'abord, on a notre mot-clé *lambda* suivi de la liste des arguments, séparés par une virgule. Ici, on en a qu'un : c'est *x*. Ensuite, un nouveau signe deux points ":" et l'instruction de la lambda. C'est l'instruction que vous mettrez ici qui sera retournée par la fonction. Dans notre exemple, on retourne donc *x* * *x*.

 Comment fait-on pour appeler notre lambda ?

On a bien créé une fonction lambda mais on ne dispose ici d'aucun moyen pour l'appeler. Vous pouvez tout simplement stocker votre fonction lambda nouvellement définie dans une variable, par une simple affectation :

Code : Python Console

```
>>> f = lambda x: x * x
>>> f(5)
25
>>> f(-18)
324
>>>
```

Un autre exemple : si vous voulez créer une fonction lambda prenant deux paramètres et retournant la somme de ces deux paramètres, la syntaxe sera la suivante :

Code : Python

```
lambda x, y: x + y
```

Utilisation

A notre niveau, les fonctions lambda sont plus une curiosité que véritablement utiles. Je vous les montre maintenant parce que c'est à propos et parce que vous pourrez en trouver certaines et ne pas comprendre ce que c'est.

Il vous faudra cependant attendre un peu pour que je vous montre une réelle application des lambda. En attendant, n'oubliez pas ce mot-clé et la syntaxe qui va avec... on passe à la suite !

À la découverte des modules

Jusqu'ici, nous avons travaillé avec les fonctions de Python chargées au lancement de l'interpréteur. Il y en a déjà un certain nombre, et nous pourrions continuer et finir cette première partie sans utiliser de module Python... ou presque. Mais il faut bien que je vous montre cette possibilité des plus intéressantes à un moment donné !

Les modules, qu'est-ce que c'est ?

Un module est grossièrement un bout de code que l'on a enfermé dans un fichier. On emprisonne ainsi des fonctions, des variables, toutes ayant un rapport entre elles. Ainsi, si l'on veut travailler avec les fonctionnalités prévues par le module (celles qui ont été enfermées dans le module), vous n'avez qu'à **importer** le module et utiliser ensuite toutes les fonctions et variables prévues.

Il existe un grand nombre de modules disponibles avec Python sans avoir besoin d'installer de bibliothèques supplémentaires. Pour cette partie, nous prendrons l'exemple du module math qui contient, comme son nom l'indique, des fonctions mathématiques. Inutile de vous inquiéter, nous n'allons pas nous attarder sur le module-même pour coder une calculatrice scientifique, nous verrons surtout les différentes méthodes d'importation .

La méthode `import`

En ouvrant l'interpréteur Python, les fonctionnalités du module *math* ne sont pas incluses. Il s'agit en effet d'un module, il vous appartient de l'importer si vous vous dites "tiens, mon programme risque d'avoir besoin de fonctions mathématiques". Nous allons voir une première syntaxe d'importation.

Code : Python Console

```
>>> import math
>>>
```

La syntaxe est facile à retenir : le mot-clé `import` qui signifie "importer" en anglais, et le nom du module, ici *math*.

Après l'exécution de cette instruction, rien ne se passe... en apparence. En réalité, Python vient d'importer le module *math*. Toutes les fonctions mathématiques contenues dans ce module sont maintenant accessibles. Pour appeler une fonction du module, il faut taper le nom du module, un point "." puis le nom de la fonction. C'est la même syntaxe pour appeler des variables du module. Voyons un exemple :

Code : Python Console

```
>>> math.sqrt(16)
4
>>>
```

Comme vous le voyez, la fonction *sqrt* du module *math* retourne la racine carrée du nombre passé en paramètre.

Mais, comment je suis censé savoir quelles fonctions existent, et que fait math.sqrt dans ce cas précis ?

J'aurais dû vous montrer cette fonction bien plus tôt, car oui, c'est une fonction qui va nous donner la solution. Il s'agit de `help`, qui prend en paramètre la fonction ou le module sur lequel vous demandez de l'aide. L'aide est fournie en anglais, mais c'est de l'anglais technique, c'est-à-dire une forme de l'anglais que vous devrez maîtriser pour programmer, si ce n'est pas le cas. Une grande majorité de la documentation est en anglais, bien que vous puissiez en trouver pas mal en français maintenant.

Code : Python Console

```
>>> help("math")
Help on built-in module math:

NAME
math

FILE
(built-in)

DESCRIPTION
This module is always available. It provides access to the
mathematical functions defined by the C standard.

FUNCTIONS
acos(...)
acos(x)

Return the arc cosine (measured in radians) of x.

acosh(...)
acosh(x)

Return the hyperbolic arc cosine (measured in radians) of x.

asin(...)
-- Suite --
```

Si vous parlez un minimum l'anglais, vous avez accès à une description exhaustive des fonctions du module *math*. Vous voyez en haut de la page le nom du module, le fichier le contenant, puis la description du module. Ensuite se trouve une liste des fonctions avec une courte description de chacune.

Tapez "q" pour revenir à la fenêtre d'interpréteur, la touche "espace" pour avancer d'une page, la touche "entrer" pour avancer d'une ligne. Vous pouvez également passer un nom de fonction en paramètre de la fonction `help`.

Code : Python Console

```
>>> help("math.sqrt")
Help on built-in function sqrt in module math:

sqrt(...)
sqrt(x)

Return the square root of x.

>>>
```

Ne mettez pas les parenthèses habituelles après le nom de la fonction. C'est en réalité la référence de la fonction que vous envoyez à `help`. Si vous rajoutez les parenthèses ouvrantes et fermantes après le nom de la fonction, vous devrez préciser une valeur et, si vous le faites, c'est la valeur retournée par `math.sqrt` qui sera analysée, soit un nombre (entier ou flottant).

Nous reviendrons plus tard sur le concept des références des fonctions. Si vous avez compris pourquoi il ne fallait pas mettre de parenthèses après le nom de la fonction dans `help`, tant mieux. Sinon, ce n'est pas grave, nous y reviendrons en temps voulu.

Utiliser un espace de noms spécifique

En vérité, quand vous tapez **`import math`**, un espace de noms se crée, portant le nom math, contenant les variables et fonctions du module "math". Quand vous tapez `math.sqrt(25)`, vous précisez à Python que vous souhaitez exécuter la fonction `sqrt` contenue dans l'espace de noms `math`. Cela signifie que vous pouvez avoir, dans l'espace de noms principal, une autre fonction sqrt que vous avez défini. Il n'y aura pas de conflit entre la fonction que vous avez créée et que vous appellerez grâce à l'instruction `sqrt` et la fonction *sqrt* du module math que vous appellerez grâce à l'instruction `math.sqrt`.

Mais concrètement, un espace de noms c'est quoi ?

Il s'agit de regrouper certaines fonctions et variables avec un préfixe spécifique. Prenons un exemple concret :

Code : Python

```
import math
a = 5
b = 33.2
```

- Dans l'espace de noms principal, celui qui ne nécessite pas de préfixe et que vous utilisez depuis le début de ce tutoriel, on trouve :
  - La variable a
  - La variable b
  - Le module *math* qui se trouve dans un espace de nom s'appelant *math* également. Dans cet espace de nom, on trouve :
    - La fonction sqrt
    - La variable pi
    - Et bien d'autres fonctions et variables...

C'est aussi l'intérêt des modules : des variables et fonctions sont stockées à part, bien à l'abri dans un espace de noms, sans risque de conflit avec vos propres variables et fonctions. Mais dans certains cas, vous pourrez vouloir changer le nom de l'espace de noms dans lequel le module importé sera stocké.

Code : Python

```
import math as mathematiques
mathematiques.sqrt(25)
```

Qu'est-ce qu'on a fait là ?

On a simplement importé le module *math* en lui spécifiant qu'il ne devrait pas être contenu dans l'espace de noms *math*, mais *mathematiques*. Cela permet simplement de contrôler un peu mieux les espaces de nom des modules que vous importerez. Dans la plupart des cas, vous n'utiliserez pas cette fonctionnalité j'imagine, mais au moins vous savez qu'elle existe. Quand on se

penchera sur les packages, vous vous souviendrez probablement de cette possibilité.

Une autre méthode d'importation : `from ... import ...`

Il existe une autre méthode d'importation qui ne fonctionne pas tout à fait de la même façon. En fonction du résultat attendu, j'utilise l'une ou l'autre méthode indifféremment. Si nous reprenons notre exemple du module *math*. Admettons que nous avons uniquement besoin dans notre programme de la fonction retournant la valeur absolue d'une variable. Dans ce cas, nous n'allons importer que la fonction, au lieu d'importer tout le module.

Code : Python Console

```
>>> from math import fabs
>>> fabs(-5)
5
>>> fabs(2)
2
>>>
```

Pour ceux qui n'ont pas encore étudié les valeurs absolues, il s'agit tout simplement de l'opposé de la variable si elle est négative, et de la variable elle-même si elle est positive. Une valeur absolue est ainsi toujours positive.

Vous aurez remarqué qu'on ne met plus le préfixe `math.` devant le nom de la fonction. En effet, nous l'avons importé avec la méthode **`from`** qui charge la fonction depuis le module indiqué et le place dans l'interpréteur au même plan que les fonctions existantes de l'interpréteur, tout comme **`print`**. Si vous avez compris les explications sur les espaces de noms, **`print`** et `fabs` sont dans le même espace de noms (principal).

Vous pouvez appeler toutes les variables et fonctions d'un module en tapant "*" à la place du nom de la fonction à importer.

Code : Python Console

```
>>> from math import *
>>> sqrt(4)
2
>>> fabs(5)
5
```

A la ligne 1 de notre programme, l'interpréteur a parcouru toutes les fonctions et variables du module *math* et les a importées directement dans l'espace de noms principal sans les emprisonner dans l'espace de noms *math*.

Bilan

Quelle méthode faut-il utiliser ?

Vaste question ! Je dirais que c'est à vous de voir. La seconde méthode a l'avantage inestimable que l'on n'a pas besoin de taper le nom du module avant d'utiliser ses fonctions. L'inconvénient de cette méthode est que si on utilise plusieurs modules de cette manière, et que par hasard, deux fonctions du même nom existent dans deux modules différents, l'interpréteur gardera la dernière fonction appelée (je vous rappelle qu'il ne peut y avoir deux variables ou fonctions du même nom). Conclusion... c'est à vous de voir en fonction de vos besoins  .

Allez, prenez bien le temps de souffler. Dans le chapitre suivant, on va bien moins travailler, je vais surtout vous apprendre à mettre vos programmes Python dans des fichiers et à créer des modules (comme vous le verrez, ça va de pair)  . Quand vous êtes prêts, en avant ! 

Pas à pas vers la modularité (2/2)

Nous allons commencer par voir comment mettre nos programmes en boîte... ou plutôt en fichier. Je vais faire d'une pierre deux coups : d'abord, c'est chouette d'avoir son programme dans un fichier éditable, surtout qu'on commence à pouvoir faire des programmes assez sympas, même si vous n'en avez peut-être pas l'impression. Et ensuite, c'est un prélude nécessaire à la création de modules. C'est parti !

Mettre en boîte notre code
Fini, l'interpréteur ?

Je le répète encore, l'interpréteur est véritablement très pratique pour un grand nombre de raisons. Et la meilleure d'entre elles est qu'il propose une manière interactive d'écrire un programme avec la possibilité de tester à chaque instruction le résultat. Toutefois, l'interpréteur a aussi un défaut : le code que vous entrez est effacé à la fermeture de la fenêtre. Or, nous commençons à être capable de rédiger des programmes relativement complexes, même si vous n'en avez peut-être pas l'impression. Dans ces conditions, devoir réécrire le code entier de son programme à chaque fois qu'on ouvre l'interpréteur de commandes est assez lourd.

La solution ? Mettre notre code dans un fichier que nous pourrons lancer à volonté, comme un véritable programme !

Comme je l'ai dit dans le début de cette partie, il est grand temps que je vous montre cette possibilité. Mais on ne dit pas adieu à l'interpréteur de commandes pour autant. On lui dit juste au revoir pour cette fois... mais on le retrouvera bien assez tôt, la possibilité de tester notre code à la volée est vraiment un atout pour apprendre le langage.

Emprisonnons notre programme dans un fichier

Pour cette démonstration, je reprendrai le code optimisé du programme calculant si une année est bissextile. C'est un petit programme dont l'utilité est certes discutable mais il remplit un but précis, en l'occurrence dire si l'année entrée par l'utilisateur est bissextile ou non, et ça suffit pour un premier essai.

Je vous remet le code ici pour qu'on travaille tous sur les mêmes lignes, même si votre version marchera également sans problème dans un fichier, si elle tournait sous l'interpréteur de commandes.

Code : Python

```
# programme testant si une année, entrée par l'utilisateur,
# est bissextile ou non

print("Entrez une année :")
annee = input() # on attend que l'utilisateur entre l'année qu'il
désire tester
annee = int(annee) # risque d'erreur si l'utilisateur n'a pas rentré
un nombre

if annee%400==0 or (annee%4==0 and annee%100!=0):
    print("L'année entrée est bissextile.")
else:
    print("L'année entrée n'est pas bissextile.")
```

Ca va être à vous de travailler maintenant, je vais vous donner des pistes mais je ne vais pas me mettre à votre place, chacun prend ses habitudes en fonction de ses préférences .

Ouvrez un éditeur basique, le bloc-notes Windows est candidat, Wordpad ou Word sont exclus. Sous Linux, vous pouvez utiliser vim ou emacs. Copiez et collez le code dans ce fichier. Enregistrez-le en précisant l'extension .py (exemple bissextile.py). Cela permettra au système d'exploitation de savoir qu'il doit utiliser Python pour exécuter ce programme (sous Windows, sous Linux ce n'est pas nécessaire).

Sous Linux, vous devrez rajouter dans votre fichier une ligne tout au début spécifiant le chemin de l'interpréteur Python (si vous avez déjà scripté en bash par exemple cette méthode ne vous surprendra pas). La première ligne de votre programme sera :

Code : Python

```
#!adresse
```

Remplacez adresse par l'adresse ou l'on peut trouver l'interpréteur, exemple : */usr/bin/python3.2*. Vous devrez changer le droit d'exécution du fichier avant de l'exécuter comme un script.

Sous Windows, rendez-vous dans le dossier où vous avez enregistré votre fichier .py. Vous pouvez double-cliquer dessus, Windows saura qu'il doit appeler Python, grâce à l'extension .py, et Python reprend la main. Attendez toutefois car il reste quelques petites choses à régler avant de pouvoir exécuter votre programme.

Quelques ajustements

Quand on exécute un programme directement dans un fichier et que le programme contient des accents (et c'est le cas ici), il est nécessaire de préciser à Python l'encodage des accents. Je ne vais pas rentrer dans les détails, je vais simplement vous donner une ligne de code qu'il faudra mettre tout en haut de votre programme (sous Linux, juste en-dessous du chemin de l'interpréteur Python).

Code : Python

```
# -*-coding:ENCODAGE -*
```

Sous Windows, vous devrez probablement remplacer "ENCODAGE" par "Latin-1". Sous Linux, ce sera plus vraisemblablement "utf-8". Il n'est pas temps de faire un cours sur les encodages. Utilisez simplement la ligne qui marche chez vous sans oublier de la mettre et tout ira bien.

Il est probable, si vous exécutez votre application d'un double-clic, que votre programme se referme immédiatement après vous avoir demandé l'année. En fait, il fait bel et bien le calcul s'il n'y a pas eu d'erreur, mais il arrive à la fin du programme en une fraction de seconde et referme l'application, puisqu'elle est finie. Pour palier cette difficulté, il faut demander à votre programme de se mettre en pause à la fin de son exécution. Vous devrez rajouter une instruction un peu spéciale, un appel système qui marche sous Windows (pas sous Linux). Il faut tout d'abord importer le module os. Ensuite, on rajoute l'appel à la fonction `os.system` en lui passant en paramètre la chaîne de caractère "pause" (cela, à la fin de votre programme). Sous Linux, vous pouvez simplement exécuter votre programme dans la console ou, si vous tenez à faire une pause, utiliser par exemple *input* avant la fin de votre programme (pas bien élégant toutefois).

Voici donc, enfin, le code de votre premier programme l'encodage est à changer en fonction de votre système).

Code : Python

```
# -*-coding:Latin-1 -*

import os # on importe le module os qui dispose de variables et de
fonctions
          # utiles pour dialoguer avec votre système d'exploitation

# programme testant si une année, entrée par l'utilisateur,
# est bissextile ou non

print("Entrez une année :")
annee = input() # on attend que l'utilisateur entre l'année qu'il
désire tester
annee = int(annee) # risque d'erreur si l'utilisateur n'a pas rentré
un nombre
                   # si l'année est bissextile ou non
if annee%400==0 or (annee%4==0 and annee%100!=0):
    print("L'année entrée est bissextile.")
else:
    print("L'année entrée n'est pas bissextile.")

# on met le programme en pause pour éviter qu'il ne se referme
(Windows)
os.system("pause")
```

Quand vous exécutez ce script, que ce soit sous Windows ou Linux, vous faites toujours appel à l'interpréteur Python ! Votre programme n'est pas compilé, mais chaque ligne d'instruction est exécutée à la volée par l'interpréteur, le même que celui qui exécutait vos premiers programmes dans l'interpréteur de commandes. La grande différence ici est que Python exécute votre programme depuis le fichier et que donc, si vous souhaitez le modifier, il faudra modifier le fichier.

Sachez qu'il existe des éditeurs spécialisés pour Python, notamment Idle installé en même temps que Python, que personnellement je n'utilise pas. Vous pouvez l'ouvrir avec un clic droit sur votre fichier .py et regarder comment ça marche, ce n'est pas bien compliqué et vous avez la possibilité d'exécuter votre programme depuis ce logiciel. Mais étant donné que je ne l'utilise pas, je ne vous ferai pas un cours dessus. Si vous avez du mal à utiliser une des fonctionnalités du logiciel, recherchez sur Internet, d'autres tutoriels doivent exister, en anglais dans le pire des cas.

Je viens pour conquérir le monde... et créer mes propres modules

Mes modules à moi

Bon, nous avons vu le plus dur... ça va ? Rassurez-vous, nous n'allons rien faire de compliqué dans cette dernière partie. Le plus dur est derrière nous.

Commencez par vous créer un espace de test pour les petits programmes Python que nous allons être amenés à faire, un joli dossier à l'écart de vos photos et musiques . Nous allons créer deux fichiers .py dans ce dossier :

- Un fichier multipli.py qui contiendra notre fonction `table` que nous avons codé au début de ce chapitre
- Un fichier test.py qui contiendra le test d'exécution de notre module

Vous devriez vous en tirer sans problème. N'oubliez pas de spécifier la ligne contenant l'encodage en tête de vos deux fichiers. Maintenant, voyons le code du fichier multipli.py.

Code : Python

```
"""module multipli contenant la fonction table"""

def table(nb, max=10):
    """Fonction affichant la table de multiplication par nb de
1 * nb jusqu'à max * nb
"""
    i = 0
    while i<max:
        print(i+1 , "*" , nb , "=" , (i+1)*nb)
        i += 1
```

On se contente de définir une seule fonction, table, qui affiche la table de multiplication choisie. Rien de nouveau jusqu'ici. Si vous vous souvenez des docstrings, au chapitre précédent, nous en avons mis une nouvelle ici, non pas pour commenter une fonction mais bien un module entier. C'est une bonne habitude à prendre quand nos projets deviennent important.

Voici le code du fichier test.py, n'oubliez pas la ligne de votre encodage en haut .

Code : Python

```
import os
from multipli import *

# test de la fonction table
table(3, 20)
os.system("pause")
```

En le lançant directement voilà ce qu'on obtient :

Code : Console

```
1 * 3 = 3
2 * 3 = 6
3 * 3 = 9
4 * 3 = 12
5 * 3 = 15
6 * 3 = 18
7 * 3 = 21
8 * 3 = 24
9 * 3 = 27
10 * 3 = 30
11 * 3 = 33
12 * 3 = 36
13 * 3 = 39
14 * 3 = 42
15 * 3 = 45
16 * 3 = 48
17 * 3 = 51
18 * 3 = 54
19 * 3 = 57
20 * 3 = 60
Appuyez sur une touche pour continuer...
```

Je ne pense pas avoir grand chose à ajouter. Nous avons vu comment créer un module, il suffit de le mettre dans un fichier. On peut alors l'importer depuis un autre fichier **contenu dans le même répertoire** en précisant le nom du fichier (sans l'extension .py). Notre code encore une fois n'est pas très utile mais vous pouvez le modifier pour le rendre plus intéressant, vous en avez parfaitement les compétences à présent  .

Au moment d'importer votre module, Python va lire (ou créer si il n'existe pas) un fichier *.pyc*. A partir de la version 3.2, ce fichier se trouve dans un dossier *__pycache__*.

Ce fichier est généré par Python et contient le code compilé (ou presque) de votre module. Il ne s'agit pas réellement de langage machine, mais d'un format que Python décode un peu plus vite que le code que vous pouvez écrire. Python se charge lui-même de générer ce fichier et vous n'avez pas vraiment besoin de vous en soucier quand vous codez, ne soyez juste pas surpris.

Faire un test de module dans le module-même

Dans l'exemple que nous venons de voir, nous avons créé deux fichiers, le premier contenant un module, le second testant ledit module. Mais on peut très facilement tester le code d'un module dans le module-même. Cela veut dire que vous pourriez exécuter votre module comme un programme à lui tout seul, un programme qui testerait le module écrit dans le même fichier. Voyons voir cela  .

Reprenons le code du module "multipli" :

Code : Python

```
"""module multipli contenant la fonction table"""

def table(nb, max=10):
    """Fonction affichant la table de multiplication par nb de
1 * nb jusqu'à max * nb
"""
    i = 0
    while i<max:
        print(i+1 , "*" , nb , "=" , (i+1)*nb)
        i += 1
```

Ce module définit une seule fonction, "table", qu'il pourrait être bon de tester. Oui mais... si nous rajoutons une ligne comme par exemple `table(8)` juste en-dessous, cette ligne sera exécutée lors de l'importation et donc, dans le programme appelant le module. Quand vous ferez **`import multipli`**, vous verrez la table de multiplication par 8 s'afficher... hum, il y a mieux .

Heureusement, il y a un moyen très rapide de séparer le code qui sera exécuté si on lance le module directement en tant que programme, ou si on l'importe. Je vais vous livrer la solution que je détaillerai au-dessous :

Code : Python

```
"""module multipli contenant la fonction table"""

import os

def table(nb, max=10):
    """Fonction affichant la table de multiplication par nb de
1 * nb jusqu'à max * nb
"""
    i = 0
    while i<max:
        print(i+1 , "*" , nb , "=" , (i+1)*nb)
        i += 1


# test de la fonction table
if __name__ == "__main__":
    table(4)
    os.system("pause")
```

N'oubliez pas la ligne indiquant l'encodage !

Voilà. A présent, si vous double-cliquez directement sur le fichier "multipli.py", vous allez voir la table de multiplication par 4. En revanche, si vous l'importez, le code de test ne s'exécutera pas. Tout repose en fait sur la variable __name__, c'est une variable qui existe dès le lancement de l'interpréteur. Si elle vaut "__main__", cela veut dire que le fichier appelé est le fichier exécuté. Autrement dit, si __name__ est égale à "__main__", vous pouvez mettre un code qui sera exécuté si le fichier est lancé directement comme un exécutable.

Prenez le temps de comprendre ce mécanisme, faites des tests si nécessaire, ça pourra vous être utile par la suite .

Les packages

Les modules sont un des moyens de regrouper plusieurs fonctions (et comme on le verra plus tard, certaines classes également). On peut aller encore au-delà en regroupant nos modules dans ce qu'on va appeler des **packages**.

En théorie

Comme je l'ai dit, un package sert à regrouper plusieurs modules. Cela permet de ranger plus proprement vos modules, classes et fonctions dans des emplacements séparés. Si vous voulez y accéder, vous allez devoir entrer un chemin vers le module que vous visez. De ce fait, les risques de conflits de noms sont moins importants et surtout, tout est bien plus ordonné.

Par exemple, si vous installez un jour une bibliothèque tierce, disons pour écrire une interface graphique. En s'installant, la librairie ne va pas créer ses dizaines (voire ses centaines) de modules au même endroit. Ce serait un peu désordonné... surtout quand on pense qu'on peut ranger tout ça d'une façon plus claire : d'un côté, on peut avoir les différents objets graphiques de la fenêtre, de l'autre les différents événements (clavier, souris, ...), de l'autre des effets graphiques...

Dans ce cas, on va sûrement se retrouver face à un package portant le nom de la librairie. Dans ce package se trouveront probablement d'autres packages, un nommé *evenements*, un autre *objets*, un autre encore *effets*. Dans chacun de ces packages, on pourra trouver soit d'autres packages, soit des modules et dans chacun de ces modules, des fonctions.

Ouf ! Ca nous fait une hiérarchie assez complexe non ? D'un autre côté, c'est tout l'intérêt. Concrètement pour utiliser notre bibliothèque, on est pas obligé de connaître tous ses packages, modules et fonctions (heureusement d'ailleurs) mais juste ceux dont on a réellement besoin.

En pratique

En pratique, les packages sont... des répertoires ! Dedans peuvent se trouver d'autres répertoires (d'autres packages) ou des fichiers (des modules).

Exemple de hiérarchie

Pour notre bibliothèque imaginaire, la hiérarchie des répertoires et fichiers ressemblerai à cela :

- Un répertoire du nom de la librairie contenant :
  - Un répertoire *evenements* contenant :
    - Un module *clavier*
    - Un module *souris*
    - ...
  - Un répertoire *effets* contenant différents effets graphiques
  - Un répertoire *objets* contenant les différents objets graphiques de notre fenêtre (boutons, zones de texte, barres de menu...)

Importer des packages

Si vous voulez utiliser, dans votre programme, la bibliothèque fictive que nous venons de voir, vous avez plusieurs moyens, tous tournant autour des mot-clés *from* et *import* :

Code : Python

```
import nom_bibliotheque
```

Cette ligne va importer le package contenant la bibliothèque. Pour accéder aux sous-packages, vous utiliserez un point "." pour modéliser le chemin menant au module ou à la fonction que vous voulez utiliser :

Code : Python

```
nom_bibliotheque.evenements # pointe sur le sous-package evenements
nom_bibliotheque.evenements.clavier # pointe sur le module clavier
```

Si vous voulez n'importer qu'un module (ou qu'une seule fonction) d'un package, vous utiliserez une syntaxe identique, assez intuitive :

Code : Python

```
from nom_bibliotheque.objets import bouton
```

En fonction des besoins, vous pouvez décider d'importer tout un package, un sous-package, un sous-sous-package... ou bien juste un module ou même juste une fonction. Cela dépendra de vos besoins  .

Créer ses propres packages

Si vous voulez créer vos propres packages, commencez par créer, dans le même dossier que votre programme Python, un répertoire portant le nom du package.

Dedans, vous devrez créer un fichier *__init__.py* pour que Python reconnaisse ce répertoire comme un package. Ce fichier peut être vide, je vous montrerai brièvement plus loin quoi mettre dedans.

Dans ce répertoire, vous pouvez soit :

- Mettre vos modules, vos fichiers à l'extension *.py*
- Créer des sous-packages de la même façon, en créant un répertoire dans votre package et un fichier *__init__.py* dedans

Ne mettez pas d'espaces dans vos noms de package et évitez également les caractères spéciaux. Quand vous les utiliserez dans votre programme, les noms donné seront identiques à des noms de variables et obéissent donc aux mêmes règles de nommage.

Dans votre fichier *__init__.py*, vous pouvez écrire du code qui sera exécuté quand vous importerez votre package. Je ne vais pas vous donner d'exemple concret ici, vous avez déjà bien des choses à retenir  .

Un dernier exemple

Un dernier exemple que vous pouvez cette fois faire en même temps que moi pour vous assurer que ça marche  .

Dans votre répertoire de code, là où vous mettez vos exemples Python, créez un fichier *.py* que vous appelerez *test_package.py* par exemple.

Créez dans le même répertoire un dossier *package*. Dedans, créez un fichier *__init__.py* que vous laisserez vide, et un fichier *fonctions.py* dans lequel vous recopierez votre fonction *table*.

Dans votre fichier *test_package.py*, si vous voulez importer votre fonction *table*, vous avez plusieurs solutions :

Code : Python

```
from package.fonctions import table
table(5) # appel de la fonction table

# ou ...
import package.fonctions
package.fonctions.table(5) # appel de la fonction table
```

Voilà. Il reste bien des choses à dire sur les packages, mais je crois que vous avez vu l'essentiel. Cette petite explication révélera son importance quand vous aurez à construire des programmes assez importants. Evitez de tout mettre en module sans les hiérarchiser, profitez de cette possibilité offerte par Python  .
Ouf ! Et bien, ce fut long tout ça  . Heureusement, c'est fini. Vous coupez au QCM sur cette partie. Il nous reste un dernier chapitre théorique sur les exceptions, puis notre premier TP, et c'en sera fini de cette première partie introductive sur Python  .

Les exceptions

Dans ce chapitre, nous aborderons le dernier concept que je considère comme indispensable avant d'attaquer la partie sur la Programmation Orientée Objet. J'ai longtemps hésité avant de placer ce chapitre ici, et je vous préviens donc d'ores et déjà : nous aurons l'occasion de compléter ce cours sur les exceptions, notamment quand nous aborderons le sujet des objets.

À quoi ça sert ?

Nous avons déjà été confrontés à des erreurs dans nos programmes ; certaines que j'ai volontairement provoquées, mais la plupart que vous avez dû rencontrer si vous avez testé un minimum des instructions dans l'interpréteur. Quand Python rencontre une erreur dans votre code, il **lève une exception**. Sans le savoir, vous avez donc déjà vu des exceptions levées par Python :

Code : Python Console

```
>>> # exemple classique : test d'une division par zéro
>>> variable = 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: int division or modulo by zero
```

Attardons-nous sur la dernière ligne. Nous y trouvons deux informations :

- `ZeroDivisionError` : le type de l'exception
- `int division or modulo by zero` : le message qu'envoie Python pour vous aider à comprendre l'erreur qui vient de se produire.

Python lève donc des exceptions quand il trouve une erreur, soit dans le code (une erreur de syntaxe par exemple), soit dans l'opération que vous lui demandez de faire.

Notez qu'à l'instar des variables, on trouve des types d'exceptions que Python va utiliser dans plusieurs situations. Le type d'exceptions *ValueError* notamment sera levé par Python dans plusieurs situations différentes incluant des erreurs de "valeurs". C'est donc dans ce cas le message qui vous indique plus clairement le problème. Nous verrons dans la prochaine partie consacrée à la Programmation Orientée Objet ce que sont réellement ces types d'exceptions.

Bon, c'est très joli d'avoir cette exception. On voit le fichier et la ligne à laquelle s'est produite l'erreur (très pratique quand on commence à travailler sur un projet) et on a une indication sur le problème qui suffit en général à le régler. Mais Python permet quelque chose de bien plus pratique.

Admettons que certaines erreurs puissent être provoquées par l'utilisateur. Par exemple, on demande à l'utilisateur d'entrer, au clavier, un entier, et il entre une chaîne de caractères... problème. Nous avons déjà rencontré cette situation, souvenez-vous du programme *bissextile*.

Code : Python

```
annee = input() # on demande à l'utilisateur d'entrer l'année
annee = int(annee) # on essaye de convertir l'année en un entier
```

Je vous avais dit que si l'utilisateur rentrait ici quelque chose d'inconvertible en entier (une lettre par exemple), le programme planterait. En fait, il lève une exception, et Python arrête l'exécution du programme. Si vous testez le programme directement dans l'explorateur, il va se fermer tout de suite (en fait, il affiche bel et bien l'erreur mais se referme aussitôt).

Dans ce cas, et dans d'autres similaires, Python permet de tester un bout de code. S'il ne renvoie aucune erreur, Python continue. Sinon, on peut lui demander d'exécuter une autre action (par exemple, redemander à l'utilisateur d'entrer l'année). C'est ce que nous allons voir ici.

Forme minimale du bloc try

On va parler ici de **bloc *try***. Nous allons en effet mettre les instructions que nous souhaitons tester dans un premier bloc d'instructions et les instructions à exécuter en cas d'erreur dans un autre bloc d'instructions. Sans plus attendre, voici la syntaxe :

Code : Python

```
try:
    # bloc à essayer
except:
    # bloc qui sera exécuté en cas d'erreur
```

Dans l'ordre, nous trouvons :

- Le mot-clé *try* suivi des deux points ":" (try signifie "essayer" en anglais)
- Le bloc d'instructions à essayer
- Le mot-clé *except* suivi, une fois encore, des deux points ":". Il se trouve au même niveau d'indentation que le *try*
- Le bloc d'instructions qui sera exécuté si une erreur est trouvée dans le premier bloc.

Reprenons notre test de conversion en enfermant l'instruction qui pourrait lever une exception dans un bloc *try*.

Code : Python

```
annee = input()
try: # on essaye de convertir l'année en entier
    annee = int(annee)
except:
    print("Erreur lors de la conversion de l'année.")
```

Vous pouvez tester ce code en précisant plusieurs valeurs différentes à la variable *annee* comme *"2010"* ou *"annee2010"*.

J'ai parlé de **forme minimale** et ce n'est pas pour rien. D'abord il va de soi que vous ne pouvez intégrer cette solution directement dans votre code. En effet, si l'utilisateur entre une année inconvertible, le système affiche certes une erreur mais finit par planter (puisque l'année, au final, n'a pas été convertie). Une des solutions envisageable est d'attribuer une valeur par défaut à l'année en cas d'erreur, ou de redemander à l'utilisateur d'entrer l'année.

Ensuite et surtout, cette méthode est assez grossière. Elle essaye une instruction et intercepte **n'importe quelle** exception liée à cette instruction. Ici, c'est acceptable car nous n'avons pas énormément d'erreurs possibles sur cette instruction. Mais c'est une mauvaise habitude à prendre. Voici une manière plus élégante et moins dangereuse.

Forme plus complète

Nous allons apprendre à compléter notre bloc *try*. Comme je l'ai indiqué plus haut, la forme minimale est à éviter pour plusieurs raisons.

D'abord, elle ne différencie pas les exceptions qui pourront être levées dans le bloc *try*. Ensuite, Python peut lever des exceptions qui ne signifient pas nécessairement qu'il y a eu une erreur.

Exécuter le bloc *except* pour un type d'exceptions précis

Dans l'exemple que nous avons vu plus haut, on ne pense qu'à un type d'exceptions qui pourrait être levé : le type *ValueError* qui pourrait trahir une erreur de conversion. Voyons un autre exemple :

Code : Python

```
try:
    resultat = numerateur / denominateur
except:
    print("Une erreur est survenue... laquelle ?")
```

Ici, on peut trouver plusieurs erreurs possibles, levant chacune une exception différente.

- *NameError* : l'une des variables *numerateur* ou *denominateur* n'a pas été définie (elle n'existe pas). Si vous essayez dans l'interpréteur l'instruction `print(numerateur)` alors que vous n'avez pas défini la variable *numerateur*, vous aurez la même erreur
- *TypeError* : l'une des variables *numerateur* ou *denominateur* ne peut diviser ou être divisée (les chaînes de caractères ne peuvent être divisées, ni diviser d'autres types, par exemple). Cette exception est levée car vous utilisez l'opérateur de division "/" sur des types qui ne savent pas quoi en faire
- *ZeroDivisionError* : encore elle ! Si *denominateur* est égale à 0, cette exception sera levée.

Cette énumération n'est pas une liste exhaustive de toutes les exceptions qui peuvent être levées à l'exécution de ce code. Elle est surtout là pour vous montrer que plusieurs erreurs peuvent se produire sur une instruction (c'est encore plus flagrant sur un bloc constitué de plusieurs instructions) et que la forme minimale intercepte toutes ces erreurs sans les distinguer, ce qui peut être problématique parfois. 

Tout se joue sur la ligne du *except*. Entre ce mot-clé et les deux points, vous pouvez préciser le type de l'exception que vous souhaitez traiter.

Code : Python

```
try:
    resultat = numerateur / denominateur
except NameError:
    print("La variable numerateur ou denominateur n'a pas été définie.")
```

Ce code ne traite que le cas où une exception *NameError* est levée. On peut intercepter les autres types d'exceptions en faisant d'autres blocs *except* à la suite :

Code : Python

```
try:
    resultat = numerateur / denominateur
except NameError:
    print("La variable numerateur ou denominateur n'a pas été définie.")
except TypeError:
    print("La variable numerateur ou denominateur possède un type incompatible avec la division.")
except ZeroDivisionError:
    print("La variable denominateur est égale à 0.")
```

C'est mieux non ? 

Allez un petit dernier... euh... pour l'instant  .

On peut capturer l'exception et afficher son message grâce au mot-clé *as* que vous avez déjà vu, dans un autre contexte (si si, rappelez-vous de l'importation de modules ).

Code : Python

```
try:
    # bloc de test
except type de l exception as exception retournee:
```

```
        print("Voici l'erreur :", exception_retournee)
```

Dans ce cas, une variable *exception_retournee* est créée par Python si une exception du type précisé est levée dans le bloc *try*.

Je vous conseille de toujours préciser un type d'exceptions après *except* (sans nécessairement capturer l'exception dans une variable, bien entendu). D'abord, vous ne devez pas utiliser *try* comme une méthode miracle pour tester n'importe quel bout de code. Il est important que vous gardiez le maximum de contrôle sur votre code. Cela signifie que si une erreur se produit, vous devez être capable de l'anticiper. En pratique, vous n'irez pas jusqu'à tester si une variable quelconque existe bel et bien, il faut faire un minimum confiance à son code. Mais si vous êtes en face d'une division et que le dénominateur pourrait avoir une valeur de 0, mettez la division dans un bloc *try* et précisez après le *except* le type de l'exception qui risque de se produire (*ZeroDivisionError* dans cet exemple).

Si vous adoptez la forme minimale (à savoir *except* sans préciser un type d'exceptions qui pourrait se produire sur le bloc *try*), toutes les exceptions seront traitées de la même façon. Et même si "exception = erreur" la plupart du temps, ce n'est pas toujours le cas. Par exemple, Python lève une exception quand vous voulez fermer votre programme avec le raccourci CTRL + C. Ici vous ne voyez peut-être pas le problème, mais si votre bloc *try* est dans une boucle par exemple, vous ne pourrez pas arrêter votre programme avec CTRL + C, puisque l'exception sera traitée par votre *except*.

Je vous conseille donc de toujours préciser un type d'exceptions possible après votre *except*. Vous pouvez bien entendu tester dans l'interpréteur de commandes Python pour reproduire l'exception que vous voulez traiter et ainsi connaitre son type.

Les mots-clés else et finally

Ce sont deux mots-clés qui vont nous permettre de construire un bloc *try* plus complet.

else

Vous avez déjà vu ce mot-clé et j'espère que vous vous en rappelez. Dans un bloc *try*, *else* va permettre d'exécuter une action si aucune erreur n'est trouvée dans le bloc. Voici un petit exemple :

Code : Python

```
try:
    resultat = numerateur / denominateur
except NameError     :
    print("La variable numerateur ou denominateur n'a pas été
définie.")
except TypeError:
    print("La variable numerateur ou denominateur possède un type
incompatible avec la division.")
except ZeroDivisionError:
    print("La variable denominateur est égale à 0.")
else:
    print("Le résultat obtenu est", resultat)
```

Dans les faits, on utilise assez peu *else*. La plupart des codeurs préfère mettre la ligne contenant le *print* directement dans le bloc *try*. Pour ma part, je trouve que c'est important de distinguer entre le bloc *try* et ce qui s'effectue ensuite. La ligne du *print* ne produira vraisemblablement aucune erreur, inutile de la mettre dans le bloc *try*.

finally

finally permet d'exécuter du code après un bloc *try*, **quelle que soit l'exécution du dit bloc**. La syntaxe est des plus simples :

Code : Python

```
try:
    # test d'instruction(s)
except TypeDInstruction:
    # traitement en cas d'erreur
```

```
finally:
    # instruction(s) exécutée(s) qu'il y ait eu erreurs ou non
```

Est-ce que ça ne revient pas au même si on met du code juste après le bloc ?

Pas tout à fait. Le bloc *finally* est exécuté dans tous les cas de figure. Quand bien même Python trouverait une instruction *return* dans votre bloc *except* par exemple, il exécutera le bloc *finally*.

Un petit bonus : le mot-clé pass

Il peut arriver, dans certains cas, que l'on souhaite tester un bloc d'instructions... mais ne rien faire en cas d'erreur. Toutefois, un bloc *try* ne peut être seul.

Code : Python Console

```
>>> try:
...     1/0
...
  File "<stdin>", line 3

^
SyntaxError: invalid syntax
```

Il existe un petit mot-clé que l'on peut utiliser dans ce cas. Son nom est *pass* et sa syntaxe est très simple d'utilisation :

Code : Python

```
try:
    # test d'instruction(s)
except type_de_l_exception: # rien ne doit se passer en cas
d'erreur
    pass
```

Je ne vous encourage pas particulièrement à utiliser ce mot-clé mais il existe et vous le savez à présent  .

pass n'est pas un mot-clé propre aux exceptions : on peut également le trouver dans des conditions, ou dans des fonctions que l'on souhaite laisser vide.

Voilà, nous avons vu l'essentiel. Il nous reste à faire un petit point sur les assertions et à voir comment lever une exception (ce sera très rapide) et on pourra passer au QCM.

Les assertions

Les assertions sont un moyen simple de s'assurer qu'une condition est respectée avant de continuer. En général, on les utilise dans des blocs *try ... except*.

Voyons comment ça marche : nous allons pour l'occasion découvrir un nouveau mot-clé (encore un), *assert*. Sa syntaxe est la suivante :

Code : Python

```
assert test
```

Si le test renvoie *True*, l'exécution se poursuit normalement. Sinon, une exception *AssertionError* est levée.

Voyons un exemple :

Code : Python

```
>>> var = 5
>>> assert var == 5
>>> assert var == 8
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
>>>
```

Comme vous le voyez, la ligne 2 s'exécute sans problème et ne lève aucune exception. On test en effet si *var == 5*. C'est le cas, le test est donc vrai, aucune exception n'est levée.

A la ligne suivante cependant, le test est *var == 8*. Cette fois, le test est faux et une exception du type *AssertionError* est levée.


A quoi ça sert, concrètement ?

Dans notre programme testant si une année est bissextile, on pourrait vouloir s'assurer que l'utilisateur n'entre pas une année inférieure ou égale à 0 par exemple. Avec les assertions, c'est très facile à faire :

Code : Python

```
annee = input("Entrez une année supérieure à 0 :")
try:
    annee = int(annee) # convertion de l'année
    assert annee > 0
except ValueError:
    print("Vous n'avez pas entré un nombre.")
except AssertionError:
    print("L'année entrée est inférieure ou égale à 0.")
```

Lever une exception

Hmmm... je vois d'ici les mines sceptiques (non non, ne vous cachez pas !)  . Vous vous demandez probablement pourquoi vous feriez le boulot de Python en levant des exceptions. Après tout, votre boulot, c'est en théorie d'éviter que votre programme plante. 

Parfois cependant, il pourra être utile de lever des exceptions. Vous verrez tout l'intérêt du concept quand vous créerez vos propres classes... mais ce n'est pas pour tout de suite  . En attendant, je vais vous donner la syntaxe et vous pourrez faire quelques tests, vous verrez de toute façon qu'il n'y a rien de compliqué.

On utilisera un nouveau mot-clé pour lever une exception... le mot-clé *raise*.

Code : Python

```
raise TypeDeLException("message à afficher")
```

Prenons un petit exemple, toujours autour de notre programme *bissextile*. Nous allons lever une exception de type *ValueError* si l'utilisateur entre une année négative ou nulle.

Code : Python

```
annee = input() # l'utilisateur entre l'année
try:
    annee = int(annee) # on tente de convertir l'année
    if annee<=0:
        raise ValueError("l'année entrée est négative ou nulle")
except ValueError:
    print("La valeur entrée est invalide (l'année est peut-être
négative).")
```

Ce que nous venons de faire, nous aurions pu le faire sans utiliser d'exceptions, mais c'était surtout pour vous montrer la syntaxe dans un véritable contexte. Ici, on lève une exception que l'on intercepte immédiatement ou presque, l'intérêt est donc limité, mais bien entendu la plupart du temps ce n'est pas le cas.

Il reste des choses à découvrir sur les exceptions, mais on en a assez fait pour ce chapitre et cette partie.  Je ne vous demande pas de connaitre toutes les exceptions que Python est amené à utiliser (certaines d'entre elles pourront d'ailleurs n'exister que dans certains modules). En revanche, vous devez être capable de savoir, grâce à l'interpréteur de commandes, quelles exceptions peuvent être levées par Python dans une certaine situation.
Voici qui conclut ce chapitre sur les exceptions et la fin de la partie introductive sur Python approche... Il ne reste devant vous qu'un TP pour vous remettre en mémoire un peu tout ce qu'on a fait, et surtout, vous rendre compte du chemin parcouru.

Alors... à l'assaut ? 

TP 1 : tous au ZCasino

L'heure de vérité a sonné ! C'est dans ce chapitre que je vais faire montre de ma cruauté sans limite en vous lachant dans la nature... ou presque.

Ce n'est pas tout à fait votre premier TP dans le sens où le programme du chapitre 4, sur les conditions, constituait votre première expérience en la matière, mais à ce moment nous n'avions pas fait un programme très... récréatif, disons .

Si, durant la réalisation du TP, vous sentez que certaines connaissances vous manquent, revenez en arrière, prenez tout votre temps, on n'est pas pressé .

Notre sujet

Dans ce chapitre, nous allons essayer de faire un petit programme que nous appellerons ZCasino. Il s'agira d'un petit jeu de roulette très simplifié dans lequel vous pourrez jouer une certaine somme et gagner... ou perdre de l'argent (telle est la fortune, au casino). Quand vous n'avez plus d'argent, vous avez perdu.

Notre règle du jeu

Bon, la roulette, c'est très sympathique comme jeu, mais un peu trop compliqué pour un premier TP. Alors, on va simplifier les règles et je vous présente tout de suite ce que l'on obtient :

- Le joueur mise sur un numéro entre 0 et 49 (50 numéros en tout). En choisissant son numéro, il dépose la somme qu'il souhaite miser sur ce numéro.
- La roulette est constituée de 50 cases allant naturellement de 0 à 49. Les numéros pairs sont de couleur noire, les numéros impairs sont de couleur rouge. Le croupier lance la roulette, lache la bille et, quand la roulette s'arrête, relève le numéro de la case dans laquelle la bille s'est arrêtée. Dans notre programme, tous ces détails "matériels" ne seront pas repris mais ces explications sont aussi à l'intention de ceux qui ont eu la chance d'éviter les salles de casino jusqu'ici . Le numéro sur lequel s'est arrêté la bille est, naturellement, le numéro gagnant.
- Si le numéro gagnant est celui sur lequel le joueur a misé (probabilité de 1/50, plutôt faible), le croupier lui remet 3 fois la somme misée.
- Sinon, le croupier regarde si le numéro misé par le joueur est de la même couleur que le numéro gagnant (s'ils sont tous les deux pairs ou tous les deux impairs). Si c'est le cas, le croupier lui remet 50 % de la somme misée. Si ce n'est pas le cas, le joueur perd sa mise.

Dans les deux scénarios gagnants vus au-dessus (le numéro misé et le numéro gagnant sont identiques ou ont la même couleur), le croupier remet au joueur sa somme avant d'y ajouter ses gains. Cela veut dire que dans ces deux scénarios il récupère de l'argent. Il n'y a que dans le troisième cas qu'il perd la somme misée.

N.B. : on utilisera pour devise le dollar $ à la place de l'euro pour des raisons d'encodage sous la console Windows.

Organisons notre projet

Bon pour ce projet, nous n'allons pas écrire de module. Nous allons utiliser ceux de Python qui sont bien suffisants pour l'instant, notamment celui pour générer de l'aléatoire que je vais présenter plus bas. En attendant, ne vous privez quand même pas de faire un répertoire et d'y mettre le fichier *ZCasino.py*, tout va se jouer à l'intérieur.

Vous êtes capable d'écrire le programme *ZCasino* tel qu'expliqué dans la première partie sans difficulté... sauf pour générer de l'aléatoire. Python a dédié tout un module à la génération de pseudo-aléatoire, le module *random*.

Le module random

Nous allons nous intéresser tout particulièrement à la fonction *randrange* de ce module qui peut s'utiliser de deux manières.

- En ne précisant qu'un paramètre (`randrange(6)`) : retourne un nombre aléatoire compris entre 0 et 5
- En précisant deux paramètres (`randrange(1, 7)`) : retourne un nombre aléatoire compris entre 1 et 6 (utile pour reproduire une expérience avec un dé à six faces).

Pour tirer un nombre aléatoire compris entre 0 et 49 et simuler l'expérience du jeu de la roulette, nous allons donc utiliser l'instruction `randrange(50)`.

Il existe d'autres façons d'utiliser *randrange* mais nous n'en aurons pas besoin ici et je dirais même que pour ce programme, seule la première utilisation vous sera utile.

N'hésitez pas à faire des tests dans l'interpréteur de commande (vous n'avez pas oublié où c'est hein ?) et essayez plusieurs syntaxes de la fonction *randrange*. Je vous rappelle qu'elle se trouve dans le module *random*, n'oubliez pas de l'importer .

Arrondir un nombre

Vous l'avez peut-être bien noté, dans l'explication des règles je spécifiais que si le joueur misait sur la bonne couleur, il obtenait 50% de sa mise. Oui mais... c'est quand même mieux de travailler avec des entiers. Si le joueur mise 3 $ par exemple, on lui rend 1,5 $. C'est encore acceptable mais si ça se poursuit, on va vraiment arriver à des nombres flottants avec beaucoup de chiffres après la virgule. Alors, autant arrondir le nombre au supérieur. Ainsi, si le joueur mise 3$, on lui rend 2$. Pour cela, on va utiliser une fonction du module *math* nommée *ceil*. Je vous laisse regarder ce qu'elle fait, il n'y a rien de compliqué.

A vous de jouer

Voilà, vous avez toutes les clés en main pour coder ce programme. Prenez le temps qu'il faut pour y arriver, ne vous ruez pas sur la correction, le but du TP est que vous appreniez à coder vous-même un programme... et celui-ci n'est pas très difficile. Si vous avez du mal, morcelez le programme, ne codez pas tout d'un coup. Et n'hésitez pas à passer par l'interpréteur pour tester des fonctionnalités, c'est réellement une chance qui vous est donnée, ne la laissez pas passer .

A vous de jouer !

Correction !

C'est encore une fois l'heure de comparer nos versions. Et une fois encore, il est très peu probable que vous ayez un code identique au mien. Donc si le vôtre marche, je dirais que c'est l'essentiel. Si vous vous heurtez à des difficultés insurmontables, le forum est là pour poser vos questions.

... ATTENTION... voici... la solution !

Code : Python

```
# ce fichier abrite le code du ZCasino, un jeu de roulette adapté

import os
from random import randrange
from math import ceil

# déclaration des variables de départ
argent = 1000 # on a 1000 $ au début du jeu
continuer_partie = True # booléen qui est vrai tant qu'on doit
continuer la
                        # partie

print("Vous vous installez à la table de roulette avec", argent,
"$.")

while continuer_partie: # tant qu'on doit continuer la partie
    # on demande à l'utilisateur d'entrer le nombre surlequel il va
miser
    nombre_mise = -1
    while nombre_mise<0 or nombre_mise>49:
        print("Entrez le nombre sur lequel vous voulez miser (entre
0 et 49) :")
        nombre_mise = input()
        # on convertit le nombre misé
        try:
            nombre_mise = int(nombre_mise)
        except ValueError:
            print("Vous n'avez pas entré de nombre")
            nombre_mise = -1
            continue
```

```
            if nombre_mise<0:
                print("Ce nombre est négatif")
            if nombre_mise>49:
                print("Ce nombre est supérieur à 49")

        # à présent, on sélectionne la somme à miser sur le nombre
        mise = 0
        while mise<=0 or mise>argent:
            print("Entrez le montant de votre mise :")
            mise = input()
            # on converti la mise
            try:
                mise = int(mise)
            except ValueError:
                print("Vous n'avez pas entré de nombre")
                mise = -1
                continue
            if mise<=0:
                print("La mise entrée est négative ou nulle.")
            if mise>argent:
                print("Vous ne pouvez miser autant, vous n'avez que",
argent, "$")

        # le nombre misé et la mise ont été sélectionnés par
l'utilisateur
        # on fait tourner la roulette
        numero_gagnant = randrange(50)
        print("La roulette tourne... ... et s'arrête sur le numéro",
numero_gagnant)

        # on va établir le gain du joueur
        if numero_gagnant == nombre_mise:
            print("Félicitations ! Vous obtenez", mise * 3, "$ !")
            argent += mise * 3
        elif numero_gagnant %2 == nombre_mise %2: # ils sont de la même
couleur
            mise = ceil(mise * 0.5)
            print("Vous avez misé sur la bonne couleur. Vous obtenez",
mise, "$")
            argent += mise
        else:
            print("Désolé l'ami, c'est pas pour cette fois. Vous perdez
votre mise.")
            argent -= mise

        # on va interrompre la partie si le joueur est ruiné
        if argent<=0:
            print("Vous êtes ruiné ! C'est la fin de la partie.")
            continuer_partie = False
        else:
            # on affiche l'argent du joueur
            print("Vous avez à présent", argent, "$")
            print("Souhaitez-vous quitter le casino (o/n) ?")
            quitter = input()
            if quitter=="o" or quitter=="O":
                print("Vous quittez le casino avec vos gains.")
                continuer_partie = False

    # on met en pause le système (Windows)
    os.system("pause")
```

Encore une fois, n'oubliez pas la ligne spécifiant l'encodage si vous voulez éviter les surprises  .

Une petite chose qui pourrait vous surprendre est la construction des boucles pour tester si le joueur a entré une valeur correcte (quand on demande à l'utilisateur d'entrer un nombre entre 0 et 49 par exemple, il faut s'assurer qu'il l'a bien fait). C'est assez simple en vérité : on attend que le joueur entre un nombre. Si le nombre n'est pas valide, on demande à nouveau au joueur d'entrer ce nombre. J'en ai profité pour utiliser le concept des exceptions afin de s'assurer que l'utilisateur entrait bien un nombre.

Comme vous l'avez vu, si ce n'est pas le cas, on affiche un message d'erreur. La valeur de la variable qui contiendra le nombre est remise à *-1* (c'est-à-dire une valeur qui indiquera à la boucle que nous n'avons toujours pas eu de valeur valide de l'utilisateur) et on utilise le mot-clé *continue* pour passer les autres instructions du bloc (sans quoi, vous auriez un autre message qui s'afficherait, disant que le nombre entré est négatif... c'est plus pratique ainsi) . De cette façon, si l'utilisateur entre une donnée inconvertible, le jeu ne plante pas et lui redemande tout simplement d'entrer une valeur valide.

La boucle principale tourne autour d'un booléen. On utilise une variable *continuer_partie* qui est "vrai" tant qu'on doit continuer la partie. Une fois que la partie doit s'interrompre, elle passe à "faux". Notre boucle globale qui gère le déroulement de la partie travaille sur ce booléen et donc, dès qu'il est "faux", la boucle s'interrompt et le programme se met en pause. Tout le reste, vous devriez le comprendre sans aide, les commentaires sont là pour vous aider. Si vous avez des doutes, vous pouvez tester les lignes d'instructions problématiques dans votre interpréteur de commandes Python, encore une fois n'oubliez pas cet outil.

Et maintenant ?

Prenez bien le temps de lire ma version mais surtout de modifier la vôtre, si vous êtes arrivé à une version qui marche bien, ou qui marche presque. Ne mettez pas ce projet à la corbeille sous prétexte que nous avons fini de le coder et qu'il marche. On peut toujours améliorer un projet, et celui-ci ne fait évidemment pas exception. Vous trouverez probablement de nouveaux concepts, dans la suite de ce tutoriel, qui pourront être utilisés dans le programme de ZCasino .

Voilà qui est fait. Ce TP conclut le chapitre introductif sur Python. Il vous reste bien des choses à apprendre sur ce langage et ça commence dès la prochaine partie, consacrée à la programmation orientée objet... un nom mystérieux cachant un concept fabuleux. Nous allons surtout découvrir de nouveaux types de données plus complexes et intéressants que ceux que nous avons déjà vus (à savoir les listes, dictionnaires, fichiers...). C'est parti !

Et bien c'en est fini des concepts de base. Dès la prochaine partie, on s'attaque à la POO, la Programmation Orientée Objet, un concept franchement fascinant et très puissant en Python. Vous allez surtout apprendre à manier de nouveaux types de données, notamment les listes, les dictionnaires, les fichiers... ça donne envie non ?

Partie 2 : La Programmation Orientée Objet en tant qu'utilisateur

Vous croyiez avoir tout vu ? Surprise ! Enfin j'espère pas trop grande...

Nous avons encore beaucoup de choses à voir. La première partie de ce cours nous a permis d'approcher les concepts de base de Python et de réaliser nos premiers programmes. Les deux parties qui suivent seront consacrées à l'Orienté Objet, une méthode de programmation très riche !

Dans la partie courante, nous aborderons l'objet du point de vue de l'utilisateur. Nous allons apprendre à nous servir plus à fond des chaînes de caractères ; nous aborderons les listes, les dictionnaires et les fichiers. Dans la partie suivante, vous pourrez enfin créer vos propres objets .

Mais qu'est-ce qu'un objet ?

Avant tout, je vais répondre à cette question, et on entre dans le vif du sujet tout de suite avec le premier chapitre.

Notre premier objet : les chaînes de caractères

Les objets... vaste sujet ! Et avant d'en créer, on va d'abord voir ce dont il s'agit, par l'exemple. On va commencer avec les chaînes de caractères, un type que vous pensez bien connaître .

Vous avez dit objet ?

La première question qui risque de vous empêcher de dormir si je n'y réponds pas tout de suite, c'est :

Mais c'est quoi un objet ?

Eh bien j'ai lu beaucoup de définitions très différentes et je n'ai pas trouvé de points communs à toutes ces définitions. Nous allons donc partir d'une définition incomplète, mais qui suffira pour l'instant :

Un objet est une structure de données, comme les variables, qui peut contenir elle-même d'autres variables et fonctions. On étoffera plus loin cette définition, elle suffit bien pour le moment .

Je ne comprends rien. Passe encore qu'une variable en contienne d'autres, après tout les chaînes de caractères contiennent bien des caractères, mais qu'une variable contienne des fonctions... ça rime à quoi ?

Je pourrais passer des heures à expliquer la théorie du concept que vous n'en seriez pas beaucoup plus avancé. J'ai choisi de vous montrer les objets par l'exemple et donc, vous allez très rapidement voir ce que tout cela signifie. Mais vous allez devoir me faire confiance au début sur l'utilité de la méthode objet .

Avant d'attaquer, une petite précision. J'ai dit qu'un objet était un peu comme une variable... en fait, pour être plus exact, il faut dire qu'une variable est un objet . Toutes les variables avec lesquelles nous avons travaillé jusqu'ici sont des objets. Les fonctions que nous avons vues sont également des objets. **Tout est objet en Python**, gardez-le à l'esprit.

Assez de théorie ! Je vais expliquer tout ça tout de suite et, si je fais bien mon travail, vous devriez comprendre tout ça très rapidement .

Les méthodes de la classe str

Ohlala, j'en vois qui grimacent rien qu'en voyant le titre . Pas de raison de s'inquiéter ! On va aller tout doucement.

Je vais vous poser un problème : comment peut-on mettre une chaîne de caractères en minuscule ? Si vous vous êtes contenté de mon tutoriel, vous ne pourrez pas faire cet exercice, j'ai volontairement évité de trop aborder les chaînes de caractères jusqu'ici. Mais admettons que vous arriviez à coder une fonction prenant en paramètre la chaîne en question. Vous aurez un code qui ressemblera à cela :

Code : Python Console

```
>>> chaine = "NE CRIE PAS SI FORT !"
>>> mettre_en_minuscule(chaine)
'ne crie pas si fort !'
```

Sachez que dans les anciennes versions de Python, celles qui n'intégraient pas encore l'approche objet, on avait un module spécialisé dans le traitement des chaînes de caractères. On importait ce module et on pouvait appeler la fonction pour mettre en minuscule une chaîne. Ce module existe d'ailleurs encore et reste utilisé pour certains traitements spécifiques. Mais on va découvrir ici une façon de faire différente. Regardez attentivement :

Code : Python Console

```
>>> chaine = "NE CRIE PAS SI FORT !"
>>> chaine.lower() # mettre en minuscule la chaîne
'ne crie pas si fort !'
```

La fonction *lower* est une nouveauté pour vous. Vous devez reconnaître le point « . » qui symbolisait déjà, dans le chapitre sur les modules, une relation d'appartenance (*a.b* signifiait *b* contenu dans *a*). Ici c'est la même signification : la fonction *lower* est une fonction de la variable *chaine*.

La fonction *lower* est propre aux chaînes de caractères. Toutes les chaînes peuvent faire appel à cette fonction. Si vous tapez `type(chaine)` dans l'interpréteur, vous obtenez `<class 'str'>`. Nous avons dit qu'une variable est issue d'un type de donnée. Je vais à présent reformuler : un **objet** est issu d'une **classe**. La **classe** est une forme de type de donnée, sauf qu'elle permet de définir des fonctions et variables propres au type. C'est pour cela que dans toutes les chaînes de caractères, on peut appeler la fonction *lower*. C'est tout simplement parce que la fonction *lower* a été définie dans la **classe** *str*. Les fonctions définies dans une classe sont appelées des **méthodes**.

Récapitulons. Nous avons découvert :

- Les **objets** que j'ai présentés comme des variables, pouvant contenir d'autres variables ou fonctions (que l'on appelle **méthodes**). On appelle une méthode d'un objet grâce à `objet.methode()` .
- les **classes** que j'ai présentées comme des types de données. Une classe est un modèle qui servira à construire un objet ; c'est dans la classe qu'on va définir les méthodes propres à l'objet.

Schématiquement voici le mécanisme qui vous permet d'appeler la méthode *lower* d'une chaîne :

1. Les développeurs de Python ont créé la classe *str* qui sera utilisée pour créer des chaînes de caractères. Dans cette classe, ils ont défini plusieurs méthodes, comme *lower*, qui pourront être utilisées par n'importe quel objet construit sur cette classe.
2. Quand vous écrivez `chaine = "NE CRIE PAS SI FORT !"` , Python reconnaît qu'il doit créer une chaîne de caractères. Il va donc créer un objet d'après la classe (le modèle) qui a été définie à l'étape précédente.
3. Vous pouvez ensuite appeler toutes les méthodes de la classe *str* depuis l'objet *chaine* que vous venez de créer.

Ouf ! Ça fait beaucoup de choses nouvelles, du vocabulaire et du concept un peu particulier  .

Vous ne voyez peut-être pas encore tout l'intérêt d'avoir des méthodes définies dans une certaine classe. Cela permet d'abord de bien séparer les diverses fonctionnalités (on ne peut mettre en minuscule un nombre entier, ça n'a aucun sens). Ensuite, c'est plus intuitif, une fois passé le choc de la première rencontre  .

Bon, on parle, on parle, mais on code pas beaucoup  .

Mettre en forme une chaîne

Non, vous n'allez pas apprendre à mettre une chaîne en gras, souligné, avec une police Verdana de 15px... sachons raison garder, nous ne sommes encore que dans une console  . Nous venons de voir *lower*, il existe d'autres méthodes... mais avant tout voyons un contexte d'utilisation.

Certains d'entre vous se demandent peut-être l'intérêt de mettre des chaînes en minuscule... alors voici un petit exemple.

Code : Python

```
chaine = str() # créée une chaîne vide
               # on aurait obtenu le même résultat en tapant chaine
= ""

while chaine.lower() != "q":
    print("Tapez 'Q' pour quitter...")
    chaine = input()

print("Merci !")
```

Vous devez comprendre rapidement ce programme. On demande dans une boucle à l'utilisateur d'entrer la lettre "Q" pour quitter. Tant que l'utilisateur entre une autre lettre, la boucle continue de s'exécuter. Dès que l'utilisateur entre *Q*, la boucle s'arrête et le programme affiche `Merci !` □a devrait vous rappeler quelque chose... direction le TP de la partie 1 pour ceux qui ont la mémoire courte  .

La petite nouveauté est dans le test de la boucle : *chaine.lower() != "q"*. On prend la chaîne entrée par l'utilisateur, on la met en minuscule et on regarde si elle est différente de "q". Cela veut dire que l'utilisateur peut entrer "q" en majuscule ou en minuscule, dans les deux cas la boucle s'arrêtera.

Notez que `chaine.lower()` retourne la chaîne en minuscule, mais ne modifie pas la chaîne. Cela est très important, nous verrons pourquoi dans le chapitre suivant.

Notez aussi que nous avons appelé la fonction *str* pour créer une chaîne vide. Je ne vais pas trop compliquer les choses, pas encore, mais sachez qu'appeler ainsi un type comme une fonction permet de créer un objet de la classe. Ici, *str()* crée un objet « chaîne de caractères ». Nous avons vu dans la première partie *int()* qui crée aussi un entier (depuis un autre type si nécessaire, ce qui permet de convertir une chaîne en entier par exemple).

Bon, voyons d'autres méthodes. Je vous invite à tester mes exemples (ils sont commentés, mais on retient mieux en essayant par soi-même). 

Code : Python Console

```
>>> minuscule = "une chaine en minuscule"
>>> minuscule.upper() # mettre en majuscule
'UNE CHAINE EN MINUSCULE'
>>> minuscule.capitalize() # la première lettre en majuscule
'Une chaine en minuscule'
>>> espaces = " une chaine avec des espaces "
>>> espaces.strip() # on retire les espaces au début et à la fin de
la chaîne
'une chaine avec des espaces'
>>> titre = "introduction"
>>> titre.upper().center(20)
' INTRODUCTION '
>>>
```

La dernière instruction mérite quelques explications. Dans l'ordre :

- On appelle la méthode *upper* de l'objet *titre*. Cette méthode, comme vous l'avez vu plus haut, retourne la chaîne de caractères contenue dans l'objet, en majuscule.
- On appelle ensuite la méthode *center*, méthode que nous n'avons pas encore vue et qui permet de centrer une chaîne. On lui passe en paramètre la taille de la chaîne que l'on souhaite obtenir et la méthode, travaillant sur l'objet, va rajouter alternativement une espace au début et à la fin de la chaîne, jusqu'à obtenir la taille demandée. Dans cet exemple, *titre* contient la chaîne *'introduction'*, chaîne qui (en minuscule ou en majuscule ) mesure 12 caractères. On demande à *center* de centrer cette chaîne dans un espace de 20 caractères. La méthode *center* va donc placer 4 espaces avant le titre, et 4 espaces après, pour faire 20 en tout  .

Bon, mais maintenant, sur quel objet travaille *center* ? Sur *titre* ? Non. Sur la chaîne retournée par *titre.upper()*, c'est-à-dire le titre en majuscule. C'est pourquoi on peut chaîner ces deux méthodes : *upper*, comme la plupart des méthodes de chaînes, travaille sur une chaîne et retourne une chaîne... qui elle aussi va posséder les méthodes propres à une chaîne de caractères. Si ce n'est pas très clair, faites quelques tests, avec *titre.upper()* et *titre.center(20)*, en passant par une seconde variable si nécessaire, pour vous rendre compte du mécanisme, ce n'est pas bien compliqué.

Je n'ai mis ici que quelques méthodes, il y en a bien d'autres. Vous pouvez en voir la liste dans l'aide, en tapant, dans l'interpréteur : `help(str)` .

Formater et afficher une chaîne

Attends, on a appris à faire ça depuis cinq bons chapitres ! On va pas tout réapprendre quand même ?

Pas tout, non. Mais nous allons apprendre à considérer ce que nous savons à travers le modèle objet. Et vous allez vous rendre compte que la plupart du temps nous n'avons fait qu'effleurer les fonctionnalités du langage. Vous allez peut-être même demander grâce avant la fin de ce chapitre  .

Je ne vais pas revenir sur ce que j'ai dit, pour afficher une chaîne, on passe par la fonction *print*.

Code : Python

```
chaine = "Bonjour tout le monde !"
print(chaine)
```

Rien de nouveau ici. En revanche, on va un peu changer nos habitudes quand on désire afficher plusieurs variables.

Jusqu'ici, nous avons utilisé *print* en lui passant plusieurs paramètres. Ca marche, mais nous allons voir quelque chose légèrement plus flexible, qui d'ailleurs n'est pas seulement utile pour l'affichage.

Code : Python Console

```
>>> prenom = "Paul"
>>> nom = "Dupont"
>>> age = 21
>>> print("Je m'appelle {0} {1} et j'ai {2} ans.".format(prenom,
nom, age))
Je m'appelle Paul Dupont et j'ai 21 ans.
```

Mais ! C'est quoi ça ?

Question légitime. Voyons un peu  .

Première syntaxe de la méthode format

Nous avons utilisé une méthode de la classe *str* (encore une) pour formater notre chaîne. De gauche à droite, nous avons :

- une chaîne de caractères qui ne présente rien de particulier, sauf ces accolades entourant des nombres, d'abord *0*, puis *1*, puis *2* ;
- nous appelons la méthode *format* de cette chaîne en lui passant en paramètre les variables à afficher, dans un ordre bien précis ;
- quand Python exécute cette méthode, il va remplacer dans notre chaîne *{0}* par la première variable passée à la méthode

format (soit le prénom), *{1}* par la deuxième variable... ainsi de suite.

Souvenez-vous qu'en programmation, on commence à compter à partir de 0.

Bien, mais on aurait pu faire exactement la même chose en passant plusieurs valeurs à *print*, non ?

Absolument. Mais rappelez-vous que cette fonctionnalité est bien plus puissante qu'un simple affichage, vous pouvez formater des chaînes de cette façon. Ici, nous avons directement affiché la chaîne formatée, mais nous aurions pu la stocker :

Code : Python Console

```
>>> nouvelle_chaine = "Je m'appelle {0} {1} et j'ai {2}
ans.".format(prenom, nom, age)
>>>
```

Pour faire la même chose sans utiliser *format*, on aurait dû concaténer des chaînes, ce que nous allons voir un peu plus loin, mais ça reste plus élégant.

Dans cet exemple, nous avons appelé les variables dans l'ordre où nous les placions dans *format*, mais ce n'est pas une obligation. Considérez cet exemple :

Code : Python Console

```
>>> prenom = "Paul"
>>> nom = "Dupont"
>>> age = 21
>>> print( \
...   "Je m'appelle {0} {1} ({3} {0} pour l'administration) et j'ai
{2} ans." \
...   .format(prenom, nom, age, nom.upper()))
Je m'appelle Paul Dupont (DUPONT Paul pour l'administration) et j'ai
21 ans.'
```

J'ai coupé notre instruction, plutôt longue, à l'aide du signe « \ » placé avant un saut de ligne, pour dire à Python que l'instruction se prolongeait au-dessous.

Si vous avez du mal à comprendre l'exemple, relisez l'instruction en remplaçant vous-même les nombres entre accolades par les variables (n'oubliez pas de compter à partir de 0).

Cette première syntaxe suffit la plupart du temps, mais n'est pas forcément intuitive quand on insère beaucoup de variables : on doit retenir leur position dans l'appel de *format* pour comprendre laquelle est affichée à tel endroit. Mais il existe une autre syntaxe.

Seconde syntaxe de la méthode format

On peut également nommer les variables que l'on va afficher, c'est souvent plus intuitif que d'utiliser leur indice. Voici un nouvel exemple :

Code : Python

```
# formatage d'une adresse
adresse = """
{no_rue}, {nom_rue}
{code_postal} {nom_ville} ({pays})
""".format(no_rue=5, nom_rue="rue des Postes", code_postal=75003, \
    nom_ville="Paris", pays="France")
```

```
    print(adresse)
```

Code : Console

```
5, rue des Postes
  75003 Paris (France)
```

Je pense que vous voyez assez précisément en quoi consiste cette deuxième syntaxe de *format*. Au lieu de donner des nombres entre accolades, on spécifie des noms de variable qui doivent correspondre à ceux entrés comme mot-clé dans la méthode *format*. Je ne m'attarderai pas davantage sur ce point, je pense qu'il est assez clair comme cela.

La concaténation de chaînes

Nous allons glisser très rapidement sur le concept de concaténation, assez intuitif d'ailleurs. On cherche à regrouper deux chaînes en une, en mettant la seconde à la suite de la première. Cela se fait le plus simplement du monde :

Code : Python Console

```
>>> prenom = "Paul"
>>> message = "Bonjour"
>>> chaine_complete = message + prenom # on utilise le symbole '+'
>>>                                    # pour concaténer deux
chaînes
... print(chaine_complete) # résultat :
BonjourPaul
>>> # pas encore parfait, il manque une espace
... # qu'à cela ne tienne !
... chaine_complete = message + " " + prenom
>>> print(chaine_complete) # résultat :
Bonjour Paul
>>>
```

C'est assez clair je pense. Le signe '+' utilisé pour ajouter des nombres est ici utilisé pour **concaténer** deux chaînes. Essayons à présent de concaténer des chaînes et des nombres :

Code : Python Console

```
>>> age = 21
>>> message = "J'ai " + age + " ans."
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
>>>
```

Python se fâche tout rouge ! Certains langages auraient accepté cette syntaxe sans sourciller mais Python n'aime pas ça du tout.

Au début de la première partie, nous avons dit que Python était un langage à **typage dynamique**, ce qui signifie qu'il identifie lui-même les types de données et que les variables peuvent changer de type au cours du programme. Mais Python est aussi un langage **fortement typé**, et cela veut dire que les types de données ne sont pas là pour faire joli, on ne peut les ignorer. Ainsi, vous voulez ici ajouter une chaîne à un entier et à une autre chaîne. Python ne comprend pas : est-ce que les chaînes contiennent des nombres que je dois convertir pour les ajouter à l'entier ou est-ce que l'entier doit être converti en chaîne puis concaténé avec les autres chaînes ? Python ne sait pas. Il ne le fera pas tout seul. Mais il s'avère de bonne volonté puisqu'il suffit de lui demander de convertir l'entier pour pouvoir le concaténer aux autres chaînes.

Code : Python Console

```
>>> age = 21
>>> message = "J'ai " + str(age) + " ans."
>>> print(message)
J'ai 21 ans.
>>>
```

On appelle *str* pour convertir en une chaîne de caractères, comme nous avons appelé *int* pour convertir en entier. C'est le même mécanisme, sauf que convertir un entier en chaîne de caractères ne lèvera vraissemblablement aucune exception.

Le typage fort de Python est important. Il est un fondement de sa philosophie : j'ai tendance à considérer pour ma part qu'un langage faiblement typé (comme JavaScript qui aurait permis de concaténer ces deux chaînes et cet entier) crée des erreurs qui sont plus difficiles à repérer. Alors qu'ici, il nous suffit de convertir explicitement le type pour que Python sache ce qu'il doit faire.

Un petit extrait de la PEP 20 illustrera mes propos. Cette PEP (Python Enhancement Proposals : propositions d'amélioration de Python) reprend en phrases courtes et claires la philosophie de Python (c'est cadeau).

Citation

Explicit is better than implicit.

Courage ! On est presque à la fin. On jète un coup d'oeil au parcours et à la sélection de chaînes, et c'en sera fini de ce chapitre, riche en nouveautés... et en émotions .

Parcours et sélection de chaînes

Nous avons vu très rapidement dans la partie 1 un moyen de parcourir des chaînes. Nous allons en voir un second ici qui fonctionne par indice.

Parcours par indice

Vous devez vous en souvenir : j'ai dit qu'une chaîne de caractères était un ensemble constitué... de caractères . En fait, une chaîne de caractères est elle-même constituée de chaînes de caractères mais qui ne se composent que d'un caractère.

Accéder aux caractères d'une chaîne

Nous allons apprendre à accéder aux lettres constituant une chaîne. Par exemple, nous souhaitons sélectionner la première lettre d'une chaîne.

Code : Python Console

```
>>> chaine = "Salut les ZER0S !"
>>> chaine[0] # première lettre de la chaîne
'S'
>>> chaine[2] # troisième lettre de la chaîne
'l'
>>> chaine[-1] # dernière lettre de la chaîne
'!'
>>>
```

On précise entre crochets *[]* l'indice (la position du caractère auquel on souhaiterait accéder).

Rappelez-vous, on commence à compter à partir de 0. La première lettre est donc à l'indice 0, la deuxième à l'indice 1, la troisième à l'indice 2... On peut accéder aux lettres en partant de la fin en entrant un indice négatif. Quand vous entrez *chaine[-1]*, vous accédez ainsi à la dernière lettre de la chaîne (enfin, au dernier caractère, qui n'est pas une lettre ici).

On peut obtenir la longueur de la chaîne (le nombre de caractères qu'elle contient) grâce à la fonction *len*.

Code : Python Console

```
>>> chaine = "Salut"
>>> len(chaine)
5
>>>
```

Pourquoi ne pas avoir défini cette fonction comme une méthode de la classe *str* ? Pourquoi ne pourrait-on pas faire *chaine.len()* ?

En fait, c'est un peu le cas je vous montrerai bien plus loin. Mais, avant tout, *str* n'est qu'un exemple parmi d'autres d'ensembles (on en découvrira d'autres dans les prochains chapitres) et donc les développeurs de Python ont préféré créer une fonction qui travaillerait sur l'ensemble, plutôt qu'une méthode dans toutes ces classes.

Méthode de parcours par while

Eh bien vous en savez assez pour parcourir une chaîne grâce à la boucle *while*. Notez que dans la plupart des cas, on préférera parcourir un ensemble avec *for*, mais il est bien de savoir faire autrement, ça vous sera utile parfois.

Voici le code auquel vous pourriez arriver :

Code : Python

```
chaine = "Salut"
i = 0 # on appelle notre indice 'i' par convention
while i<len(chaine):
    print(chaine[i]) # on affiche le caractère à chaque tour de
boucle
    i += 1
```

N'oubliez pas d'incrémenter *i*, sinon vous allez avoir quelques surprises  .

Si vous essayez d'accéder à un indice qui n'existe pas (par exemple *25* alors que votre chaîne ne fait que 20 de longueur), Python lèvera une exception de type *IndexError*.

Une petite dernière chose : vous ne pouvez changer les lettres de la chaîne en utilisant les indices :

Code : Python Console

```
>>> mot = "lac"
>>> mot[0] = "b" # on veut remplacer 'l' par 'b'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>>
```

Python n'est pas content. Il ne veut pas que vous utilisiez les indices pour modifier des caractères de la chaîne. Pour ce faire, il va falloir utiliser la sélection.

Sélection de chaînes

Chouette ! On va faire du copier-coller ? 

Euh... presque .

Nous allons voir comment sélectionner une partie de la chaîne.

Si je souhaite par exemple sélectionner les deux premières lettres de la chaîne :

Code : Python

```
>>> presentation = "salut"
>>> presentation[0:2] # on sélectionne les deux premières lettres
'sa'
>>> presentation[2:len(presentation)] # on sélectionne la chaîne
sauf les deux premières lettres
'lut'
>>>
```

La sélection consiste donc à extraire une partie de la chaîne. Cette opération retourne le morceau de la chaîne sélectionné (encore une fois sans modifier la chaîne d'origine).

Sachez que l'on peut sélectionner du début de la chaîne jusqu'à un indice, et d'un indice jusqu'à la fin de la chaîne, sans le préciser comme nous l'avons fait. Python comprend très bien si on le sous-entend :

Code : Python

```
>>> presentation[:2] # du début jusqu'à la troisième lettre non
comprise
'sa'
>>> presentation[2:] # de la troisième lettre (comprise) à la fin
'lut'
>>>
```

Maintenant, nous pouvons reprendre notre exemple de tout à l'heure pour constituer une nouvelle chaîne, en remplaçant une lettre par une autre :

Code : Python Console

```
>>> mot = "lac"
>>> mot = "b" + mot[1:]
>>> print(mot)
bac
>>>
```

Voilà !

Ça reste assez peu intuitif non ?

Pour remplacer des lettres, ça paraît un peu lourd en effet. Et d'ailleurs on s'en sert assez rarement pour ça. Pour rechercher / remplacer, on a les méthodes *count*, *find* et *replace* dont je vous laisse le soin de comprendre le fonctionnement .

Aller, on peut passer au QCM, vous avez le droit à l'interpréteur pour celui-là .

Pfiou ! Voilà qui conclut un bien long chapitre, et pas très simple avec ça. Vous y avez affronté vos premiers objets et vous vous en êtes bien tiré, apparemment. Mais il n'empêche que ce chapitre contenait beaucoup de nouveautés, d'ordre conceptuelles, syntaxiques... Le chapitre suivant risque d'être au moins aussi long et cette tendance risque de se confirmer tout au long de cette partie 2. Mais passée la première rencontre avec les objets, ce que vous verrez vous surprendra moins. Inutile de vous le cacher

toutefois : ça commence juste. Alors, dès que la migraine est passée, on se retrouve au chapitre suivant, consacré aux listes et aux tuples.

Les listes et tuples (1/2)

Veni, vidi, vici. J'aurai réussi à vous faire connaître et j'espère, aimer le Python sans vous apprendre les listes . Mais allons ! Cette époque est révolue. Maintenant que nous commençons à étudier l'objet sous toutes ses formes, je ne vais pouvoir garder le secret plus longtemps : il existe des listes en Python. Pour ceux qui ne voient même pas de quoi je parle, vous allez vite vous rendre compte qu'avec les dictionnaires que nous verrons plus loin, c'est un type, ou plutôt une classe, dont on aura du mal à se passer.

Commençons !

Créons et éditons nos premières listes

D'abord c'est quoi, une liste ?

En Python, les listes sont des objets qui peuvent en contenir d'autres. C'est donc un ensemble, comme une chaîne de caractères, mais qui, au lieu de contenir des caractères, contient n'importe quel objet. Comme d'habitude, on va s'occuper du concept des listes avant de voir tout son intérêt .

Création de listes

On a deux moyens de créer des listes. Si je vous dis que la classe d'une liste s'appelle, assez logiquement, *list*, vous devriez déjà pouvoir trouver une manière de créer une liste.

Non ? ...

Vous allez vous habituer à cette syntaxe :

Code : Python Console

```
>>> ma_liste = list() # on crée une liste vide
>>> type(ma_liste)
<class 'list'>
>>> ma_liste
[]
>>>
```

Là encore, on appelle la classe comme une fonction pour **instancier** un objet de cette classe.

Quand vous affichez la liste, vous pouvez constater qu'elle est vide. Entre les crochets (qui sont les délimiteurs des listes en Python), il n'y a rien. On peut également utiliser ces crochets pour créer une liste.

Code : Python Console

```
>>> ma_liste = [] # on crée une liste vide
>>>
```

Ça revient au même, vous pouvez vérifier. Toutefois, on peut également créer une liste non vide, en lui indiquant les objets à y mettre directement à la création.

Code : Python Console

```
>>> ma_liste = [1, 2, 3, 4, 5] # une liste avec cinq objets
>>> print(ma_liste)
[1, 2, 3, 4, 5]
>>>
```

La liste que nous venons de créer compte cinq objets du type *int*. Ils sont classés par ordre croissant. Mais rien de tout cela n'est obligatoire.

- Vous pouvez faire des listes de toute longueur.
- Les listes peuvent contenir n'importe quel type d'objet.
- Les objets dans une liste peuvent être mis dans le désordre. Toutefois, la structure d'une liste fait que chaque objet a sa place et que l'ordre compte.

Code : Python Console

```
>>> ma_liste = [1, 3.5, "une chaine", []]
>>>
```

Nous avons ici créé une liste contenant quatre objets de types différents : un entier, un flottant, une chaîne de caractères et... une autre liste .

Voyons à présent comment accéder aux éléments d'une liste :

Code : Python Console

```
>>> ma_liste = ['c', 'f', 'm']
>>> ma_liste[0] # on veut accéder au premier élément de la liste
'c'
>>> ma_liste[2] # troisième élément
'm'
>>> ma_liste[1] = 'Z' # on remplace 'f' par 'Z'
>>> ma_liste
['c', 'Z', 'm']
>>>
```

Comme vous pouvez le voir, on accède aux éléments d'une liste de la même façon qu'on accède aux caractères d'une chaîne de caractères : on indique entre crochets l'indice de l'élément qui nous intéresse. Rappelez-vous que l'on commence à compter à partir de 0.

Contrairement à la classe *str*, la classe *list* vous permet de remplacer un élément par un autre. Les listes sont en effet des types dits mutables.

Ajouter des objets dans une liste

On peut trouver plusieurs méthodes, définies dans la classe *list*, pour ajouter des éléments dans une liste.

Ajouter un élément à la fin de la liste

On utilise la méthode *append* pour ajouter un élément à la fin de la liste.

Code : Python Console

```
>>> ma_liste = [1, 2, 3]
>>> ma_liste.append(56) # on ajoute 56 à la fin de la liste
>>> ma_liste
[1, 2, 3, 56]
>>>
```

Comment ça, je sais pas compter ?

C'est assez simple non ? On passe en paramètre de la méthode *append* l'objet que nous souhaitons ajouter à la fin de la liste.

Attention ! La méthode *append*, comme beaucoup de méthodes de listes, travaille directement sur l'objet et ne retourne rien.

Ceci est extrêmement important. Dans le chapitre précédent, nous avons vu que toutes les méthodes de chaînes ne modifient pas l'objet d'origine mais retournent l'objet modifié. Ici c'est le contraire : les méthodes de listes ne retournent rien mais modifient l'objet d'origine. Regardez ce code si ce n'est pas bien clair :

Code : Python Console

```
>>> chaine1 = "une petite phrase"
>>> chaine2 = chaine1.upper() # on met en majuscule chaine1
>>> chaine1 # on affiche la chaîne d'origine
'une petite phrase'
>>> # elle n'a pas été modifiée par la méthode upper
... chaine2 # on affiche chaine2
'UNE PETITE PHRASE'
>>> # c'est chaine2 qui contient la chaîne en majuscule
... # voyons pour les listes à présent
... liste1 = [1, 5.5, 18]
>>> liste2 = liste1.append(-15) # on ajoute -15 à liste1
>>> liste1 # on affiche liste1
[1, 5.5, 18, -15]
>>> # cette fois, l'appel de la méthode a modifié l'objet d'origine
(liste1)
... # voyons ce que contient liste2
... liste2
>>> # rien ? Vérifions avec print
... print(liste2)
None
>>>
```

Je vais expliquer les dernières lignes. Mais d'abord, il faut que vous fassiez bien la différence entre les méthodes de chaînes, où l'objet d'origine n'est jamais modifié mais qui retournent un nouvel objet, et les méthodes de listes, qui ne retournent rien mais modifient l'objet d'origine.

J'ai dit que les méthodes de listes ne retournent rien. On va pourtant essayer de capturer la valeur de retour dans *liste2*. Quand on essaye d'afficher la valeur de *liste2* en l'entrant directement, on n'obtient rien. Il faut l'afficher avec *print* pour savoir ce qu'elle contient : *None*. C'est l'objet vide de Python. En réalité, quand une fonction ne retourne rien, elle retourne *None*. Vous retrouverez peut-être ce mot-clé de temps à autre, ne soyez donc pas surpris.

Insérer un élément dans la liste

Nous allons passer assez rapidement sur cette seconde méthode. On peut, très simplement, insérer un objet dans une liste, à l'endroit voulu. On utilise pour cela la méthode *insert*.

Code : Python Console

```
>>> ma_liste = ['a', 'b', 'd', 'e']
>>> ma_liste.insert(2, 'c') # on insère 'c' à l'indice 2
>>> print(ma_liste)
['a', 'b', 'c', 'd', 'e']
```

Quand on demande d'insérer *'c'* à l'indice 2, la méthode va décaler les objets d'indice supérieur ou égal à 2. *'c'* va donc s'intercaler entre *'b'* et *'d'*.

Concaténation de listes

On peut également agrandir des listes en les concaténant avec une autre :

Code : Python Console

```
>>> ma_liste1 = [3, 4, 5]
>>> ma_liste2 = [8, 9, 10]
>>> ma_liste1.extend(ma_liste2) # on insère ma_liste2 à la fin de
ma_liste1
>>> print(ma_liste1)
[3, 4, 5, 8, 9, 10]
>>> ma_liste1 = [3, 4, 5]
>>> ma_liste1 + ma_liste2
[3, 4, 5, 8, 9, 10]
>>> ma_liste1 += ma_liste2 # identique à extend
>>> print(ma_liste1)
[3, 4, 5, 8, 9, 10]
>>>
```

Voici les différentes façons de concaténer des listes. Vous pouvez remarquer l'opérateur + qui concatène deux listes entre elles et retourne le résultat. On peut utiliser += assez logiquement pour étendre une liste. Cette façon de faire revient au même qu'utiliser la méthode *extend*.

Suppression d'éléments d'une liste

Nous allons voir rapidement comment supprimer des éléments d'une liste avant d'apprendre à les parcourir. Vous allez vite pouvoir constater que ça se fait assez simplement. Nous allons voir deux méthodes pour supprimer des éléments d'une liste :

- Le mot-clé *del*
- La méthode *remove*

Le mot-clé del

C'est un des mot-clés de Python, que j'aurais pu vous montrer plus tôt. Mais les applications de *del* me semblaient assez peu pratiques avant d'aborder les listes.

del (abréviation de delete) signifie "supprimer" en anglais. Son utilisation est des plus simple : `del variable_a_supprimer`. Voyons un exemple :

Code : Python Console

```
>>> variable = 34
>>> variable
34
>>> del variable
>>> variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'variable' is not defined
>>>
```

Comme vous le voyez, après l'utilisation de *del*, la variable n'existe plus. Python l'efface tout simplement. Mais on peut également utiliser *del* pour supprimer des éléments d'un ensemble, comme une liste, et c'est ce qui nous intéresse ici.

Code : Python Console

```
>>> ma liste = [-5, -2, 1, 4, 7, 10]
```

```
>>> del ma_liste[0] # on supprime le premier élément de la liste
>>> ma_liste
[-2, 1, 4, 7, 10]
>>> del ma_liste[2] # on supprime le troisième élément de la liste
>>> ma_liste
[-2, 1, 7, 10]
>>>
```

Si vous entrez `del ma_liste[0]` , le premier élément de la liste sera supprimé de *ma_liste*. Simple non ?

La méthode remove

On peut aussi supprimer des éléments de la liste grâce à la méthode *remove* qui prend en paramètre, non pas l'indice de l'élément à supprimer, mais l'élément lui-même.

Code : Python Console

```
>>> ma_liste = [31, 32, 33, 34, 35]
>>> ma_liste.remove(32)
>>> ma_liste
[31, 33, 34, 35]
>>>
```

La méthode *remove* parcourt la liste et en retire l'élément que vous lui passez en paramètre. C'est une façon de faire un peu différente et vous appliquerez *del* et *remove* dans des situations différentes.

Attention ! La méthode *remove* ne retire que la première occurrence de la valeur trouvée dans la liste.

Notez au passage que le mot-clé *del* n'est pas une méthode de liste. Il aurait pu l'être. Mais il s'agit d'une fonctionnalité de Python qu'on retrouve dans la plupart des objets conteneurs, tels que les listes que nous venons de voir, ou les dictionnaires que nous verrons plus tard. D'ailleurs, *del* sert plus généralement à supprimer, pas forcément des éléments d'un ensemble mais aussi, comme nous l'avons vu, des variables.

Nous allons à présent voir comment parcourir une liste, même si vous devez déjà avoir votre petite idée sur la question  .

Le parcours de listes

Vous avez déjà dû vous faire une idée des méthodes pour parcourir une liste. Je vais passer brièvement dessus, vous ne verrez rien de nouveau ni, je l'espère, de très surprenant :

Code : Python Console

```
>>> ma_liste = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
>>> i = 0 # notre indice pour la boucle while
>>> while i<len(ma_liste):
...     print(ma_liste[i])
...     i += 1 # on incrémente i, ne pas oublier !
...
a
b
c
d
e
f
g
h
>>> for elt in ma_liste: # elt va prendre les valeurs successives
des éléments de ma_liste
...     print(elt)
...
```

```
a
b
c
d
e
f
g
h
>>>
```

Il s'agit des mêmes méthodes de parcours que nous avons vues pour les chaînes de caractères, dans le chapitre précédent. Nous allons cependant aller un peu plus loin.

La fonction *range*

Les deux méthodes que nous venons de voir possèdent toutes deux des inconvénients :

- La méthode par *while* est plus longue à écrire, moins intuitive et elle est perméable aux boucles infinies, si l'on oublie d'incrémenter la variable servant de compteur
- La méthode par *for* se contente de parcourir la liste en capturant les éléments dans une variable ; sans qu'on puisse savoir où ils sont dans la liste.

C'est vrai dans le cas que nous venons de voir. Certains codeurs vont combiner les deux méthodes pour plus de flexibilité, mais très souvent le code obtenu est moins lisible. Heureusement, les développeurs de Python ont pensé à nous.

La fonction *range* prend au minimum un paramètre, un nombre. Elle retourne une liste contenant tous les entiers de 0 au nombre passé en paramètre moins un.

Code : Python Console

```
>>> range(5)
[0, 1, 2, 3, 4]
>>>>
```

On utilise parfois cette fonction pour parcourir une liste par la méthode *for*, mais en utilisant les indices.

Code : Python

```
for i in range(len(ma_liste)):
    print(ma_liste[i])
```

Décortiquez un peu cet exemple : on appelle la méthode *range* et on lui passe en paramètre la taille de *ma_liste*. La fonction *range* va donc retourner une liste allant de 0 à la taille de la liste moins un. La liste obtenue est donc la liste des indices de *ma_liste*. La boucle *for* travaille sur cette liste d'indice et elle va placer successivement chaque indice dans la variable *i*.

Pourquoi s'embêter à utiliser *range* ? On arrivait très bien à parcourir une liste avec `for elt in ma_liste .`

En effet. Mais dans certaines situations, que nous allons justement voir un peu plus bas, il est plus utile d'avoir les indices d'une liste que les éléments qu'elle contient. Souvenez-vous que grâce aux indices, on peut accéder aux éléments mais que l'inverse n'est pas vrai.

La fonction *enumerate*

Utiliser la fonction *range* pour parcourir une liste n'est ni très intuitif ni très élégant. Voici une autre façon de faire que je vais détailler plus bas :

Code : Python

```
>>> ma_liste = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
>>> for i,elt in enumerate(ma_liste):
...     print("A l'indice {0} se trouve {1}.".format(i, elt))
...
A l'indice 0 se trouve a.
A l'indice 1 se trouve b.
A l'indice 2 se trouve c.
A l'indice 3 se trouve d.
A l'indice 4 se trouve e.
A l'indice 5 se trouve f.
A l'indice 6 se trouve g.
A l'indice 7 se trouve h.
>>>
```

Pas de panique ! 

Nous avons ici une boucle *for* un peu surprenante. Entre *for* et *in*, nous avons deux variables, séparées par une virgule.

Regardons d'abord ce que renvoie la fonction *enumerate* :

Code : Python Console

```
>>> enumerate(ma_liste)
((0, 'a'), (1, 'b'), (2, 'c'), (3, 'd'), (4, 'e'), (5, 'f'), (6,
'g'), (7, 'h'))
>>>
```

Ça ne vous aide peut-être pas beaucoup. En fait, *enumerate* prend une liste en paramètre et retourne un *tuple*, contenant lui-même d'autres *tuples*, associant un indice à l'élément correspondant. Nous verrons les *tuples* un peu plus loin, pour l'instant sachez qu'il s'agit simplement de listes que l'on ne peut pas modifier.

Ce n'est sans doute pas encore très clair. Essayons d'afficher ça un peu mieux :

Code : Python Console

```
>>> for elt in enumerate(ma_liste):
...     print(elt)
...
(0, 'a')
(1, 'b')
(2, 'c')
(3, 'd')
(4, 'e')
(5, 'f')
(6, 'g')
(7, 'h')
>>>
```

Quand on parcourt chaque élément de notre *tuple* retourné par *enumerate*, on voit d'autres *tuples* qui contiennent deux éléments : d'abord l'indice, puis ensuite l'objet se trouvant à cet indice, dans la liste passée en paramètre de la fonction *enumerate*.

Si les parenthèses vous déconcertent trop, vous pouvez imaginer à la place des crochets, cela revient au même dans cet exemple.

Quand on utilise *enumerate*, on capture l'indice et l'élément dans deux variables distinctes. Voyons un autre exemple pour comprendre ce mécanisme :

Code : Python Console

```
>>> autre_liste = [
...     [1, 'a'],
...     [4, 'd'],
...     [7, 'g'],
...     [26, 'z'],
... ] # on a étalé la liste sur plusieurs lignes
>>> for nb,lettre in autre_liste:
...     print("La lettre {0} est la {1}e de
l'alphabet.".format(lettre, nb))
...
La lettre a est la 1e de l'alphabet.
La lettre d est la 4e de l'alphabet.
La lettre g est la 7e de l'alphabet.
La lettre z est la 26e de l'alphabet.
>>>
```

J'espère que c'est assez clair dans votre esprit. Dans le cas contraire, décomposez ces exemples, le déclic devrait se faire.

N.B. : On écrit ici la définition de la liste sur plusieurs lignes pour des raisons de lisibilité. On n'est pas obligé de mettre des anti-slashs \ en fin de ligne car, tant que Python ne trouve pas de crochet fermant la liste, il continue d'attendre sans interpréter la ligne. Vous pouvez d'ailleurs le constater avec les points qui remplacent les chevrons au début de la ligne, tant que la liste n'a pas été refermée.

Quand on travaille sur une liste que l'on parcours en même temps, on peut se retrouver face à des erreurs assez étranges, qui paraissent souvent incompréhensibles au début.

Par exemple, des exceptions *IndexError* si on tente de supprimer certains éléments d'une liste en la parcourant.

Nous verrons dans le chapitre suivant comment faire cela proprement, pour l'heure qu'il vous suffise de vous méfier d'un parcours qui modifie une liste, sa structure surtout. D'une façon générale, évitez de parcourir une liste dont la taille évolue en même temps.

Allez ! On va jeter un coup d'oeil aux tuples, pour conclure ce chapitre  .

Un petit coup d'œil aux tuples

Nous avons brièvement vu les *tuples* un peu plus haut, grâce à la fonction *enumerate*. J'avais dit alors que les *tuples* étaient des listes immutables, que l'on ne pouvait modifier. En fait, vous allez vous rendre compte que nous utilisons depuis longtemps des *tuples* sans nous en rendre compte.

Un *tuple* se définit comme une liste, sauf qu'on utilise comme délimiteur des parenthèses au lieu des crochets :

Code : Python

```
tuple_vide = ()
tuple_non_vide = (1,)
tuple_non_vide = (1, 3, 5)
```

Une petite subtilité ici : si on veut créer un *tuple* avec un unique élément, on doit quand même mettre une virgule après. Sinon, Python va automatiquement supprimer les parenthèses et on se retrouvera avec une variable lambda et non un *tuple* contenant cette variable.

Mais à quoi ça sert ?

Il est assez rare que l'on travaille directement sur des *tuples*. Ce sont après tout des types que l'on ne peut modifier. On ne peut supprimer d'éléments d'un tuple, ni en ajouter. Cela vous paraît peut-être encore assez abstrait, mais il peut être utile de travailler sur des données sans pouvoir les modifier.

Passons en attendant. Voyons plutôt les cas où nous avons utilisé des tuples sans le savoir.

Affectation multiple

Tous les cas que nous allons voir sont des cas d'affectation multiple. Vous vous souvenez ?

Code : Python Console

```
>>> a,b = 3,4
>>> a
3
>>> b
4
>>>
```

On a également utilisé cette syntaxe pour permuter deux variables. Et bien, cette syntaxe passe par des tuples qui ne sont pas déclarés explicitement. Vous pourriez écrire :

Code : Python Console

```
>>> (a,b) = (3,4)
>>>
```

Quand Python trouve plusieurs variables ou valeurs séparées par des virgules et sans délimiteur, il va les mettre dans des tuples. Dans le premier exemple, les parenthèses sont sous-entendues et Python comprend ce qu'il doit faire.

Une fonction retournant plusieurs valeurs

Nous ne l'avons pas vu jusqu'ici, mais une fonction peut retourner deux valeurs, ou même plus :

Code : Python

```
def decomposer(entier, divise_par):
    """Cette fonction retourne la partie entière et le reste de
entier / divise_par .
"""
    p_e = entier // divise_par
    reste = entier % divise_par
    return p_e, reste
```

Et on peut ensuite capturer la partie entière et le reste dans deux variables, au retour de la fonction :

Code : Python Console

```
>>> partie_entiere, reste = decomposer(20, 3)
>>> partie_entiere
6
>>> reste
2
>>>
```

Là encore, on passe par des tuples sans que ce soit indiqué explicitement à Python. Si vous essayez de faire `retour = decomposer(20, 3)` , vous allez capturer un tuple contenant deux éléments : la partie entière et le reste de 20 divisé par 3.

Nous verrons plus loin d'autres exemples de tuples, et d'autres utilisations. Je pense que ça suffit pour cette fois. En attendant, direction le QCM !
J'espère que ce petit tour d'horizon vous a plu, car on n'en a pas fini avec les listes. Dès le prochain chapitre, on attaque des concepts un peu plus difficiles, mais qu'il vous faut connaître. N'hésitez pas à faire une petite pause entre ces chapitres pour être bien sûr de maîtriser les concepts présentés.

Les listes et tuples (2/2)

Les listes sont très utilisées en Python. Elles sont liées à pas mal de fonctionnalités, dont certaines plutôt complexes. Aussi, j'ai préféré scinder mon approche des listes en deux chapitres. Vous allez voir dans celui-ci quelques fonctionnalités qui ne s'appliquent qu'aux listes et aux tuples, et qui pourront vous être extrêmement utiles. Je vous conseille donc, avant tout, d'être bien à l'aise avec les listes, création, parcours, édition, suppression...

D'autre part, comme la plupart des chapitres abordés, je ne peux faire un tour d'horizon de toutes les fonctionnalités de tous les objets présentés. Je vous invite donc à lire la documentation pour une liste exhaustive des méthodes, en tapant `help(list)` .

C'est parti !

Entre chaînes et listes

Nous allons voir un moyen de transformer des chaînes en listes, et réciproquement.

Il est assez surprenant, de prime abord, d'avoir une conversion possible entre ces deux types qui sont tout de même assez différents. Mais comme on va le voir, il ne s'agit pas d'une réelle conversion. Il va être difficile de démontrer l'utilité tout de suite, mieux valent quelques exemples.

Des chaînes aux listes

Pour « convertir » une chaîne en liste, on va utiliser une méthode de chaîne, nommée *split* (« découper » en anglais). Cette méthode prend un paramètre, une autre chaîne, souvent d'un caractère, qui définit comment on va découper notre chaîne initiale.

C'est un peu compliqué et ça paraît très tordu... mais regardez plutôt :

Code : Python Console

```
>>> ma_chaine = "Bonjour à tous"
>>> ma_chaine.split(" ")
['Bonjour', 'à', 'tous']
>>>
```

On passe en paramètre de la méthode *split* une chaîne contenant une unique espace. La méthode retourne une liste contenant les trois mots de notre petite phrase. Chaque mot se trouve dans une case de la liste.

C'est assez simple en fait : quand on appelle la méthode *split*, elle va découper la chaîne en fonction du paramètre donné, ici la première case de la liste va être du début jusqu'à la première espace (non incluse), puis de la première espace à la seconde, ainsi de suite jusqu'à la fin de la chaîne.

Sachez que *split* possède un paramètre par défaut, un code qui définit les espaces, les tabulations et les sauts de ligne. Donc vous pouvez très bien faire `ma_chaine.split()` , ça revient ici au même.

Des listes aux chaînes

Voyons l'inverse à présent : si on a une liste contenant des chaînes de caractères que nous souhaitons rassembler en une seule. On utilise la méthode de chaîne *join* (joindre en anglais). Sa syntaxe d'utilisation est un peu surprenante :

Code : Python Console

```
>>> ma_liste = ['Bonjour', 'à', 'tous']
>>> " ".join(ma_liste)
'Bonjour à tous'
>>>
```

En paramètre de la méthode *join*, on passe la liste des chaînes que l'on souhaite « ressouder ». La méthode va travailler sur l'objet qui l'appelle, ici une chaîne de caractères contenant une unique espace. Elle va insérer cette chaîne entre chaque chaîne de la liste, ce qui au final nous donne la chaîne de départ, *'Bonjour à tous'*.

N'aurait-il pas été plus simple ou plus logique de faire une méthode de liste, prenant en paramètre la chaîne faisant la jonction ?

Ce choix est en effet contesté, mais je ne trancherai pas pour ma part. Le fait est que c'est cette méthode qui a été choisie, et avec un peu d'habitude on arrive à bien lire le résultat obtenu. D'ailleurs, nous allons voir comment appliquer concrètement ces deux méthodes .

Une application pratique

Admettons que nous avons un nombre flottant dont nous souhaitons afficher la partie entière et la partie flottante, mais uniquement les trois premières décimales. Autrement dit, si on a un nombre flottant tel que *3.999999999999998*, on souhaite avoir comme résultat *3.999*. D'ailleurs, ce serait plus joli si on remplaçait le point décimal par la virgule, à laquelle nous sommes plus habitués .

Là encore, je vous invite à essayer de faire ce petit exercice par vous-même. On part du principe que la valeur de retour de la fonction chargée de la pseudo-conversion est une chaîne de caractères. Voici quelques exemples d'utilisation de la fonction que vous devriez coder :

Code : Python Console

```
>>> afficher_flottant(3.99999999999998)
'3,999'
>>> afficher_flottant(1.5)
'1,5'
>>>
```

Voici la correction que je vous propose :

Secret (cliquez pour afficher)

Code : Python

```
def afficher_flottant(flottant):
    """Fonction prenant en paramètre un flottant et retournant
une chaîne
de caractères avec ce nombre tronqué. La partie flottante
doit être d'une longueur maximum de 3 caractères.

De plus, on va remplacer le point décimal par la virgule.
"""
    if type(flottant) is not float:
        raise TypeError("le paramètre attendu doit être un
flottant")
    flottant = str(flottant)
    partie_entiere, partie_flottante = flottant.split(".")
    # la partie entière n'est pas à modifier. Seule la partie
flottante
    # doit être tronquée
    return ",".join([partie_entiere, partie_flottante[:3]])
```

En s'assurant que le type passé en paramètre est bien un flottant, on assure qu'il n'y aura pas d'erreurs lors du fractionnement de la chaîne. On est sûr qu'il y aura forcément une partie entière et une partie flottante séparées par un point, même si la partie flottante n'est constituée que d'un 0. Si vous n'y êtes pas arrivé par vous-même, regardez bien cette solution, elle n'est pas forcément simple au premier coup d'oeil. On fait intervenir un certain nombre de mécanismes que vous avez vus il y a peu, tâchez de bien les comprendre.

Les listes et paramètres de fonctions

Nous allons droit vers une fonctionnalité des plus intéressantes, qui fait une partie de la puissance de Python. Nous allons étudier un cas assez particulier avant de généraliser : les fonctions avec une liste inconnue de paramètres.

Notez malgré tout que ce point est assez délicat. Si vous n'arrivez pas bien à le comprendre, laissez cette sous-partie de côté, ça ne vous pénalisera pas  .

Les fonctions dont on ne connaît pas le nombre de paramètres à l'avance

Vous devriez tout de suite penser à la fonction *print* : on lui passe une liste de paramètres que la fonction va afficher, dans l'ordre où ils sont placés, séparés par une espace (ou tout autre délimiteur choisi).

Vous n'allez peut-être pas trouver sur le moment d'applications de cette fonctionnalité, mais tôt ou tard cela arrivera. La syntaxe est tellement simple que c'en est déconcertant :

Code : Python

```
def fonction(*parametres):
```

On place une étoile * devant le nom du paramètre qui accueillera la liste des paramètres. Voyons un peu plus précisément comment cela se présente :

Code : Python Console

```
>>> def fonction_inconnue(*parametres):
...     """Test d'une fonction pouvant être appelée avec un nombre
variable
... de paramètres.
... """
...     print("J'ai reçu : {0}.".format(parametres))
...
>>> fonction_inconnue() # on appelle la fonction sans paramètre
J'ai reçu : ().
>>> fonction_inconnue(33)
J'ai reçu : (33,).
>>> fonction_inconnue('a', 'e', 'f')
J'ai reçu : ('a', 'e', 'f').
>>> var = 3.5
>>> fonction_inconnue(var, [4], "...")
J'ai reçu : (3.5, [4], '...').
>>>
```

Je pense que cela suffit. Comme vous le voyez, on peut appeler notre *fonction_inconnue* avec un nombre indéterminé de paramètres, de 0 à l'infini (enfin, théoriquement ). Le fait de préciser une étoile * devant le nom du paramètre fait que Python va placer tous les paramètres de la fonction dans un *tuple*, que l'on peut ensuite traiter comme on le souhaite.

 Et les paramètres nommés dans l'histoire ? Comment sont-ils insérés dans le *tuple* ?

Ils ne le sont pas. Si vous entrez `fonction_inconnue(couleur="rouge")` , vous allez avoir une erreur : *fonction_inconnue() got an unexpected keyword argument 'couleur'* . Nous verrons dans le chapitre suivant comment capturer ces paramètres nommés.

Vous pouvez bien entendu définir une fonction avec plusieurs paramètres que l'on doit rentrer quoi qu'il arrive, et ensuite une liste de paramètres variables :

Code : Python

```
def fonction_inconnue(nom, prenom, *commentaires):
```

Dans cet exemple de définition de fonction, vous devez impérativement préciser un nom et un prénom, et ensuite vous mettez ce que vous voulez en commentaire, aucun paramètre, un, deux... ce que vous voulez .

Si on définit une liste variable de paramètres, elle doit se trouver après la liste des paramètres standards.

Cela est évident au fond. Vous ne pouvez avoir une définition de fonction comme `def fonction_inconnue(*parametres, nom, prenom)` . En revanche, si vous souhaitez avoir des paramètres nommés, il faut les mettre après cette liste. Les paramètres nommés sont un peu une exception, puisqu'ils ne figureront de toute façon pas dans le *tuple* obtenu. Voyons par exemple la définition de la fonction *print*.

Citation

print(value, ..., sep=' ', end='\n', file=sys.stdout)

Ne nous occupons pas du dernier paramètre. Il définit le descripteur vers lequel *print* envoie ses données, par défaut c'est l'écran.

D'où viennent ces points de suspension dans les paramètres ?

En fait, il s'agit d'un affichage un peu plus agréable. Si on veut réellement avoir la définition en code Python, on retombera plutôt sur :

Code : Python

```
def print(*values, sep=' ', end='\n', file=sys.stdout):
```

Petit exercice : faire une fonction *afficher* identique à *print*, c'est-à-dire prenant un nombre indéterminé de paramètres, les affichant en les séparant à l'aide du paramètre nommé *sep* et terminant l'affichage par la variable *fin*. Notre fonction *afficher* ne comptera pas de paramètre *file*. En outre, elle devra passer par *print* pour afficher (on ne connaît pas encore d'autres façons de faire). La seule contrainte est que l'appel à *print* ne doit compter qu'un seul paramètre non nommé. Autrement dit, avant l'appel à *print*, la chaîne devra avoir été déjà formatée, prête à l'affichage.

Pour que ce soit plus clair, je vous mets la définition de la fonction, ainsi que la *docstring* que j'ai écrite :

Code : Python

```
def afficher(*parametres, sep=' ', fin='\n'):
    """Fonction chargée de reproduire le comportement de print.
Elle doit finir par faire appel à print pour afficher le résultat,
mais
les paramètres devront déjà avoir été formatés. On doit passer à
print
une unique chaîne, en lui spécifiant de ne rien mettre à la fin :

print(chaine, end='')
"""
```

Voici la solution que je vous propose :

Secret (cliquez pour afficher)

Code : Python

```
def afficher(*parametres, sep=' ', fin='\n'):
    """Fonction chargée de reproduire le comportement de print.
Elle doit finir par faire appel à print pour afficher le
résultat, mais
les paramètres devront déjà avoir été formatés. On doit passer à
print
une unique chaîne, en lui spécifiant de ne rien mettre à la fin
:

print(chaine, end='')
"""
    # les paramètres sont sous la forme d'un tuple. Or, on a
besoin de
    # les convertir. Mais on ne peut pas modifier un tuple.
    # On a plusieurs possibilités, ici je choisis de convertir
    # le tuple en liste.
    parametres = list(parametres)
    # on va commencer par convertir toutes les valeurs en chaîne,
sinon
    # on va avoir quelques problèmes lors du join
    for i,parametre in enumerate(parametres):
        parametres[i] = str(parametre)
    # la liste des paramètres ne contient plus que des chaînes de
caractères
    # à présent on va constituer la chaîne finale
    chaine = sep.join(parametres)
    # on ajoute le paramètre fin à la fin de la chaîne
    chaine += fin
    # on affiche l'ensemble
    print(chaine, end='')
```

J'espère que ce n'était pas trop difficile, et que si vous avez fait des erreurs, vous avez pu les comprendre.

Ce n'est pas du tout grave si vous avez réussi à coder cette fonction d'une manière différente. Au contraire, tant que vous comprenez la solution que je propose .

Transformer une liste en paramètres de fonction

C'est peut-être un peu moins fréquent, mais vous devez connaître ce mécanisme puisqu'il complète parfaitement le premier. Si vous avez un *tuple*, ou une liste contenant des paramètres qui doivent être passés à une fonction, vous pouvez très simplement les transformer en paramètres lors de l'appel. Le seul problème c'est que côté démonstration je me vois un peu limité.

Code : Python Console

```
>>> liste_des_parametres = [1, 4, 9, 16, 25, 36]
>>> print(*liste_des_parametres)
1 4 9 16 25 36
>>>
```

Ce n'est pas bien spectaculaire, et pourtant c'est une fonctionnalité très puissante du langage. Là, on a une liste contenant des paramètres, et on la transforme en une liste de paramètres de la fonction *print*. Donc, au lieu que ce soit la liste qui soit affichée, ce sont tous les nombres, séparés par des espaces. C'est exactement comme si vous aviez fait `print(1, 4, 9, 16, 25, 36)` .

Mais quel intérêt ? Ça ne change pas grand-chose, et il est rare que l'on capture les paramètres d'une fonction dans une liste, non ?

Oui je vous l'accorde. Ici l'intérêt ne saute pas aux yeux. Mais un peu plus tard, vous pourrez tomber sur des applications où les fonctions sont utilisées sans savoir quels paramètres elles attendent réellement. Si on ne connaît pas la fonction que l'on appelle, c'est très pratique. Là encore, vous découvrirez ça dans les chapitres suivants ou dans certains projets. Essayez de garder à l'esprit ce mécanisme de transformation.

On utilise une étoile * dans les deux cas. Si c'est dans une définition de fonction, ça signifie que les paramètres entrés non attendus lors de l'appel seront capturés dans la variable, sous la forme d'un *tuple*. Si c'est dans un appel de fonction, au contraire, cela signifie que la variable sera décomposée en plusieurs paramètres envoyés à la fonction.

J'espère que vous êtes encore en forme, on attaque le point que je considère comme le plus dur de ce chapitre, mais aussi le plus intéressant. Gardez les yeux ouverts  !

Les compréhensions de liste

Les compréhensions de liste (« *list comprehensions* » en anglais) sont un moyen de filtrer ou modifier une liste très simplement. La syntaxe est déconcertante au début, mais vous allez voir que c'est très puissant  .

Parcours simple

Les compréhensions de liste permettent de parcourir une liste en en retournant une seconde, modifiée ou filtrée. Pour l'instant, nous allons voir une simple modification :

Code : Python Console

```
>>> liste_origine = [0, 1, 2, 3, 4, 5]
>>> [nb * nb for nb in liste_origine]
[0, 1, 4, 9, 16, 25]
>>>
```

Je vous avais prévenus  .

Étudions un peu cette ligne. Comme vous avez pu le deviner, elle signifie en langage plus conventionnel « Mettre au carré tous les nombres contenus dans la liste d'origine. » Nous trouvons dans l'ordre, entre les crochets qui sont les délimiteurs d'une instruction de compréhension de liste :

- `nb * nb` : la valeur de retour. Pour l'instant, on ne sait pas ce qu'est la variable *nb*, on sait juste qu'il faut la mettre au carré. Notez qu'on aurait pu écrire `nb**2`, cela revient au même.
- `for nb in liste_origine` : voilà d'où vient notre variable *nb*. On reconnaît la syntaxe d'une boucle *for*, sauf qu'on n'est pas habitué à la voir sous cette forme.

Quand Python interprète cette ligne, il va parcourir la liste d'origine et mettre chaque élément de la liste au carré. Il retourne ensuite le résultat obtenu, sous la forme d'une liste qui est de la même longueur que celle d'origine. On peut naturellement capturer cette nouvelle liste dans une variable.

Filtrage avec un branchement conditionnel

On peut aussi filtrer une liste de cette façon :

Code : Python Console

```
>>> liste_origine = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> [nb for nb in liste_origine if nb%2==0]
[2, 4, 6, 8, 10]
>>>
```

On rajoute à la fin de l'instruction une condition qui va déterminer quelles valeurs seront transférées dans la nouvelle liste. Ici, on ne transfert que les valeurs paires. Au final, on se retrouve donc avec une liste deux fois plus petite que celle d'origine .

N.B. : si vous travaillez sur un tuple, au lieu d'une liste, vous remplacerez les crochets encadrant les compréhensions de liste par des parenthèses.

Mélangeons un peu tout ça

Il est possible de filtrer et modifier une liste assez simplement. Par exemple, on a une liste contenant les quantités de marchandises stockées pour un magasin. Prenons des fruits, par exemple (je suis pas sectaire vous pouvez prendre des hamburgers si vous préférez). Chaque semaine, le magasin va emprunter en stock une certaine quantité de chaque fruit, pour les mettre en vente. À ce moment, le stock de chaque fruit diminue naturellement. Inutile en conséquence de garder les fruits qu'on n'a plus en stock.

Je vais un peu reformuler. On va avoir une liste simple, qui contiendra des entiers, précisant la quantité de chaque fruit (c'est abstrait, les fruits ne sont pas précisés). On va faire une compréhension de liste pour diminuer d'une quantité donnée toutes les valeurs de cette liste, et en profiter pour retirer celles qui sont inférieures ou égales à 0.

Code : Python Console

```
>>> qtt_a_retirer = 7 # on retire chaque semaine 7 fruits de chaque
sorte
>>> fruits_stockes = [15, 3, 18, 21] # par exemple 15 pommes, 3
melons...
>>> [nb_fruits-qtt_a_retirer for nb_fruits in fruits_stockes if
nb_fruits>qtt_a_retirer]
[8, 11, 14]
>>>
```

Comme vous le voyez, le fruit de quantité 3 n'a pas survécu à cette semaine d'achat. Bien sûr, cet exemple n'est pas complet : on n'a aucun moyen fiable d'associer les nombres restants aux fruits. Mais vous avez un exemple de filtrage et modification d'une liste.

Prenez bien le temps de regarder ces exemples, les compréhensions de liste ne sont pas forcément simples dans leur syntaxe au début. Faites des essais, c'est aussi le meilleur moyen de comprendre.

Nouvelle application concrète

De nouveau, c'est à vous de bosser .

Nous allons en gros reprendre l'exemple précédent, en le modifiant un peu pour qu'il soit plus cohérent. Nous travaillons toujours avec des fruits, sauf que cette fois nous allons associer un nom de fruit avec la quantité restante en magasin. Nous verrons dans le chapitre suivant comment le faire avec des dictionnaires, pour l'instant on va se contenter de listes :

Code : Python Console

```
>>> inventaire = [
...     ("pomme", 22),
...     ("melon", 4),
...     ("poire", 18),
...     ("fraise", 76),
...     ("prune", 51),
... ]
>>>
```

Recopiez cette liste. Elle contient des *tuples*, contenant chacun un couple, le nom du fruit et sa quantité en magasin.

Votre mission : trier cette liste en fonction de la quantité de chaque fruit. Autrement dit, on doit obtenir quelque chose comme ça :

Code : Python

```
[
    ("fraise", 76),
    ("prune", 51),
    ("pomme", 22),
    ("poire", 18),
    ("melon", 4),
]
```

Pour ceux qui n'ont pas eu la curiosité de regarder dans la documentation des listes, je signale à votre attention la méthode *sort* qui permet de trier une liste. Vous pouvez utiliser également la fonction *sorted* qui prend en paramètre la liste à trier (ce n'est pas une méthode de liste, faites attention). *sorted* retourne la liste triée, sans modifier l'originale, ce qui peut être utile dans certaines circonstances, précisément celle-ci. À vous de voir, vous pouvez y arriver par les deux méthodes.

Bien entendu, essayez de faire cet exercice en utilisant les compréhensions de liste.

Je vous donne juste un petit indice : vous ne pouvez trier la liste comme ça, il faut l'inverser (autrement dit, placer la quantité avant le nom du fruit) pour pouvoir ensuite la trier par quantité.

Voici la correction que je vous propose :

Secret (cliquez pour afficher)

Code : Python

```
# on va placer l'inventaire dans l'autre sens, la quantité avant
le nom
inventaire_inverse = [(qtt, nom_fruit) for nom_fruit,qtt in
inventaire]
# on n'a plus qu'à trier dans l'ordre décroissant l'inventaire
inversé
# on reconstitue l'inventaire trié
inventaire = [(nom_fruit, qtt) for qtt,nom_fruit in
sorted(inventaire_inverse, \
    reverse=True)]
```

Ca marche, et le traitement a été fait en deux lignes.

Si vous trouvez ça plus compréhensible, vous pouvez trier l'inventaire inversé avant la reconstitution. Il faut privilégier la lisibilité à la quantité de lignes.

Code : Python

```
# on va placer l'inventaire dans l'autre sens, la quantité avant
le nom
inventaire_inverse = [(qtt, nom_fruit) for nom_fruit,qtt in
inventaire]
# on trie l'inventaire inversé dans l'ordre décroissant
inventaire_inverse.sort(reverse=True)
# et on reconstitue l'inventaire
inventaire = [(nom_fruit, qtt) for qtt,nom_fruit in
inventaire_inverse)]
```

Tu n'as pas dit qu'il fallait mettre des parenthèses à la place des crochets quand on travaillait sur des tuples ?

Si, absolument. Mais là on travaille sur une liste. Elle contient des *tuples*, c'est entendu, mais l'inventaire est une liste, et c'est l'inventaire que l'on parcourt grâce aux compréhensions de liste.

Faites des essais, entraînez-vous, vous en aurez sans doute besoin, la syntaxe n'est pas très simple au début. Et évitez de tomber dans l'extrême aussi : certaines opérations ne sont pas faisables avec les compréhensions de listes, ou alors sont trop condensées pour être facilement comprises. Dans l'exemple précédent, on aurait très bien pu remplacer nos deux à trois lignes d'instructions par une seule, mais ça aurait été dur à lire. Ne sacrifiez pas la lisibilité au détriment de la longueur de votre code.

Citation

Readability counts.

Ce tour d'horizon des listes et tuples est terminé. Cela ne veut pas dire que tout a été vu, bien entendu.

Quand vous êtes prêts, on attaque un autre type très utilisé, presque plus que les listes : les dictionnaires .

Les dictionnaires

Nous y voilà . Le seigneur des types, dans toute sa splendeur .

Maintenant que vous commencez à vous familiariser avec la programmation orientée objet, nous allons pouvoir aller un peu plus vite sur les manipulations "classiques" de ce type, pour nous concentrer sur quelques petites spécificités propres aux dictionnaires .

Création et édition de dictionnaires

Un dictionnaire est un type de données extrêmement puissant et pratique. Il se rapproche des listes sur certains points, mais sur beaucoup il diffère totalement. Python utilise ce type pour représenter beaucoup de fonctionnalités : on peut par exemple retrouver les attributs d'un objet grâce à un dictionnaire particulier.

Mais n'anticipons pas. Dans les deux chapitres précédents, nous avons découvert les listes. Les objets de ce type sont des objets conteneurs, dans lesquels on trouve d'autres objets. Pour accéder à ces objets contenus, il faut connaître leur position dans la liste. Cette position se traduit par des entiers, des indices, compris entre 0 (inclus) et la taille de la liste (non incluse). Tout ça vous devez déjà le savoir .

Le dictionnaire est aussi un objet conteneur. Il n'a quant à lui aucune structure ordonnée, à la différence des listes. De plus, pour accéder aux objets contenus dans le dictionnaire, on n'utilise pas nécessairement des indices mais des **clés** qui peuvent être de bien des types distincts.

Créer un dictionnaire

Là encore, je vous donne le nom de la classe sur laquelle se construit un dictionnaire : *dict*. Vous devriez du même coup trouver la première méthode d'instanciation du dictionnaire :

Code : Python Console

```
>>> mon_dictionnaire = dict()
>>> type(mon_dictionnaire)
<class 'dict'>
>>> mon_dictionnaire
{}
>>> # du coup, vous devriez trouver la deuxième manière de créer un
dictionnaire vide
... mon_dictionnaire = {}
>>> mon_dictionnaire
{}
>>>
```

Les parenthèses délimitent les *tuples*, les crochets délimitent les listes et les accolades *{}* délimitent les dictionnaires.

Voyons comment ajouter des clés et valeurs dans notre dictionnaire vide :

Code : Python Console

```
>>> mon_dictionnaire = {}
>>> mon_dictionnaire["pseudo"] = "Prolixe"
>>> mon_dictionnaire["mot de passe"] = "*"
>>> mon_dictionnaire
{'mot de passe': '*', 'pseudo': 'Prolixe'}
>>>
```

Nous indiquons entre crochets la clé à laquelle nous souhaitons accéder. Si la clé n'existe pas, elle est ajoutée au dictionnaire avec la valeur spécifiée après le signe =. Sinon, l'ancienne valeur à l'emplacement indiqué est remplacée par la nouvelle :

Code : Python Console

```
>>> mon_dictionnaire = {}
>>> mon_dictionnaire["pseudo"] = "Prolixe"
>>> mon_dictionnaire["mot de passe"] = "*"
>>> mon_dictionnaire["pseudo"] = "6pri1"
>>> mon_dictionnaire
{'mot de passe': '*', 'pseudo': '6pri1'}
>>>
```

La valeur *'Prolixe'* pointée par la clé *'pseudo'* a été remplacée, à la ligne 4, par la valeur *'6pri1'*. Cela devrait vous rappeler la création de variables : si la variable n'existe pas, elle est créée, sinon elle est remplacée par la nouvelle valeur.

Pour accéder à la valeur d'une clé précise, c'est très simple :

Code : Python Console

```
>>> mon_dictionnaire["mot de passe"]
'*'
>>>
```

Si la clé n'existe pas dans le dictionnaire, une exception de type *KeyError* sera levée.

Généralisons un peu tout ça : nous avons des dictionnaires, qui peuvent contenir d'autres objets. On place et accède à ces objets grâce à des clés. Un dictionnaire ne peut naturellement pas contenir deux clés identiques (comme on l'a vu, la seconde valeur écrase la première). En revanche, rien n'empêche d'avoir deux valeurs identiques dans le dictionnaire.

Nous avons utilisé ici, pour nos clés et nos valeurs, des chaînes de caractères. Ce n'est absolument pas obligatoire. Vous pouvez utiliser, comme des listes, des entiers comme clé :

Code : Python Console

```
>>> mon_dictionnaire = {}
>>> mon_dictionnaire[0] = "a"
>>> mon_dictionnaire[1] = "e"
>>> mon_dictionnaire[2] = "i"
>>> mon_dictionnaire[3] = "o"
>>> mon_dictionnaire[4] = "u"
>>> mon_dictionnaire[5] = "y"
>>> mon_dictionnaire
{0: 'a', 1: 'e', 2: 'i', 3: 'o', 4: 'u', 5: 'y'}
>>>
```

On a l'impression de recréer le fonctionnement d'une liste, mais ce n'est pas le cas : rappelez-vous qu'un dictionnaire n'a pas de structure ordonnée. Si vous supprimez par exemple l'indice *2*, le dictionnaire, contrairement aux listes, ne va pas décaler toutes les clés d'indice supérieur à l'indice supprimé, il n'a pas été fait pour.

On peut utiliser quasiment tous les types comme clé, et on peut utiliser absolument tous les types comme valeur.

Voici un exemple un peu plus atypique de clés : on souhaite représenter un plateau d'échecs. Traditionnellement, on représente une case de l'échiquier par une lettre (de A à H) suivie d'un chiffre (de 1 à 8). La lettre définit la colonne, le chiffre la ligne. Regardez ce schéma si vous n'êtes pas sûr de comprendre.

Pourquoi ne pas faire un dictionnaire avec, en clé un *tuple* contenant la lettre et le chiffre identifiant la case, et en valeur le nom de la pièce ?

Code : Python

```
echiquier = {}
echiquier['a', 1] = "tour blanche" # en bas à gauche de l'échiquier
```

```
echiquier['b', 1] = "cavalier blanc" # à droite de la tour
echiquier['c', 1] = "fou blanc" # à droite du cavalier
echiquier['d', 1] = "reine blanche" # à droite du fou
# ... première ligne des blancs
echiquier['a', 2] = "pion blanc" # devant la tour
echiquier['b', 2] = "pion blanc" # devant le cavalier, à droite du
pion
# ... seconde ligne des blancs
```

Dans cet exemple, nos *tuples* sont sous-entendus. On ne les place pas entre parenthèses. Python comprend qu'on veut créer des *tuples*, ce qui est bien, mais l'important est que vous le compreniez bien aussi. Certains cours encouragent à toujours placer des parenthèses autour des *tuples* quand on les utilise. Pour ma part, je pense que si vous gardez à l'esprit qu'il s'agit de *tuples*, que vous n'avez aucune peine à l'identifier, ça suffit. Si vous faites la confusion, mettez des parenthèses autour des tuples en toute circonstance.

On peut aussi créer des dictionnaires déjà remplis :

Code : Python

```
placard = {"chemise":3, "pantalon":6, "tee-shirt":7}
```

On précise entre accolades la clé, le signe deux points « : » et la valeur correspondante. On sépare les différents couples *clé:valeur* par une virgule. C'est d'ailleurs comme ça que Python vous affiche un dictionnaire quand vous lui demandez.

Certains ont peut-être essayé de créer des dictionnaires déjà remplis avant que je ne le montre. Une petite précision, si vous avez entré une instruction similaire :

Code : Python

```
mon_dictionnaire = {'pseudo', 'mot de passe'}
```

Dans cette occasion, ce n'est pas un dictionnaire que vous créez, mais un *set*.

Un *set* est un objet conteneur, lui aussi, très semblable aux listes, sauf qu'il ne peut contenir deux objets identiques. Vous ne pouvez trouver dans un *set* deux fois l'entier *3* par exemple. Je vous laisse vous renseigner sur les *sets* si vous le désirez.

Supprimer des clés d'un dictionnaire

Comme pour les listes, vous avez deux possibilités, mais elles reviennent sensiblement au même :

- le mot-clé *del* ;
- la méthode de dictionnaire *pop*.

Je ne vais pas m'attarder sur le mot-clé *del*, il fonctionne de la même façon que pour les listes :

Code : Python

```
placard = {"chemise":3, "pantalon":6, "tee shirt":7}
del placard["chemise"]
```

La méthode *pop* supprime également la clé précisée, mais elle retourne la valeur supprimée.

Code : Python Console

```
>>> placard = {"chemise":3, "pantalon":6, "tee shirt":7}
>>> placard.pop("chemise")
3
>>>
```

La méthode *pop* retourne la valeur qui a été supprimée en même temps que la clé. Ce peut être parfois utile.

Voilà pour le tour d'horizon. Ce fut bref et vous n'avez pas vu toutes les méthodes, bien entendu. Je vous laisse consulter l'aide pour une liste détaillée.

Un peu plus loin

On se sert parfois des dictionnaires pour stocker des fonctions.

Je vais juste vous montrer rapidement le mécanisme sans trop m'y attarder. Là, je compte sur vous pour faire des tests si vous êtes intéressé. C'est encore un petit quelque chose que vous n'utiliserez peut-être pas tous les jours mais qui peut être utile à connaître.

Les fonctions sont manipulables comme des variables. Ce sont des objets, un peu particuliers mais des objets tout de même. Donc on peut les prendre pour valeur d'affectation ou les ranger dans des listes ou dictionnaires. C'est pourquoi je présente cette fonctionnalité à présent, auparavant j'aurais manqué d'exemples pratiques.

Code : Python Console

```
>>> print_2 = print # l'objet print_2 pointera sur la fonction print
>>> print_2("Affichons un message")
Affichons un message
>>>
```

On copie la fonction *print* dans une autre variable *print_2*. On peut ensuite appeler *print_2* et la fonction va afficher le texte entré, tout comme *print* l'aurait fait.

En pratique, on affecte rarement des fonctions comme cela. C'est peu utile. Par contre, on met parfois des fonctions dans des dictionnaires :

Code : Python Console

```
>>> def fete():
...     print("C'est la fête.")
...
>>> def oiseau():
...     print("Fais comme l'oiseau...")
...
>>> fonctions = {}
>>> fonctions["fete"] = fete # on ne met pas les parenthèses
>>> fonctions["oiseau"] = oiseau
>>> fonctions["oiseau"]
<function oiseau at 0x00BA5198>
>>> fonctions["oiseau"]() # on essaye de l'appeler
Fais comme l'oiseau...
>>>
```

Prenons dans l'ordre si vous le voulez bien :

- On commence par définir deux fonctions, *fete* et *oiseau* (pardonnez l'exemple)
- On créée un dictionnaire nommé *fonctions*
- On met dans notre dictionnaire nos fonctions *fete* et *oiseau*. La clé pointant vers la fonction est le nom de la fonction, tout bêtement, mais on aurait pu lui donner un nom plus original
- On essaye d'accéder à notre fonction *oiseau* en entrant *fonctions["oiseau"]*. Python nous retourne un truc assez moche, *<function oiseau at 0x00BA5198>*, mais vous comprenez l'idée : c'est bel et bien notre fonction *oiseau*. Mais pour l'appeler, il faut des parenthèses, comme toute fonction qui se respecte
- En entrant `fonctions["oiseau"]()` , on accède à la fonction *oiseau* et on l'appelle dans la foulée.

On peut stocker les références des fonctions dans n'importe quel objet conteneur, des listes, des dictionnaires... et d'autres classes, quand nous apprendrons à en faire. Je ne vous demande pas de comprendre absolument la manipulation des références des fonctions, essayez de retenir cet exemple. Dans tous les cas, nous aurons l'occasion d'y revenir .

Les méthodes de parcours

Comme vous pouvez le penser, on ne parcourt pas tout à fait un dictionnaire comme une liste. La différence n'est pas si énorme que ça, mais la plupart du temps, on passe par des méthodes de dictionnaire.

Parcours des clés

Peut-être avez-vous déjà essayé par vous-même de parcourir un dictionnaire comme on l'a fait pour les listes :

Code : Python Console

```
>>> fruits = {"pomme":21, "melon":3, "poire":31}
>>> for cle in fruits:
...     print(cle)
...
melon
poire
pomme
>>>
```

Comme vous le voyez, si on essaye de parcourir un dictionnaire « simplement », on parcourt en réalité la liste des clés contenues dans le dictionnaire.

Mais... les clés ne s'affichent pas dans l'ordre dans lesquelles on les a entrées... c'est normal ?

Les dictionnaires n'ont pas de structure ordonnée, gardez-le à l'esprit. Donc en ce sens oui, c'est tout à fait normal.

Une méthode de la classe *dict* permet d'obtenir ce même résultat. Personnellement, je l'utilise plus fréquemment car on est sûr, en lisant l'instruction, que c'est la liste des clés que l'on parcourt :

Code : Python Console

```
>>> fruits = {"pomme":21, "melon":3, "poire":31}
>>> for cle in fruits.keys():
...     print(cle)
...
melon
poire
pomme
>>>
```

La méthode *keys* (« clés » en anglais) retourne la liste des clés contenues dans le dictionnaire. En vérité, ce n'est pas tout à fait une liste (essayez d'entrer dans votre interpréteur `fruits.keys()`), mais c'est un ensemble qui se parcourt tout à fait

comme une liste.

Parcours des valeurs

On peut aussi parcourir les valeurs contenues dans un dictionnaire. Pour ce faire, on utilise la méthode *values* (« valeurs » en anglais).

Code : Python Console

```
>>> fruits = {"pomme":21, "melon":3, "poire":31}
>>> for valeur in fruits.values():
...     print(valeur)
...
3
31
21
>>>
```

Cette méthode est peu utilisée pour un parcours, car il est plus pratique de parcourir la liste des clés, cela suffit pour avoir les valeurs correspondantes. Mais on peut aussi, bien entendu, l'utiliser dans une condition :

Code : Python Console

```
>>> if 21 in fruits.values():
...     print("Un des fruits se trouve dans la quantité 21.")
...
Un des fruits se trouve dans la quantité 21.
>>>
```

Parcours des clés et valeurs simultanément

Pour avoir en même temps les indices et les objets d'une liste, on utilise la fonction *enumerate*, j'espère que vous vous en souvenez. Pour faire la même chose avec les dictionnaires, on utilise la méthode *items*. Elle retourne une liste, contenant les couples *clé:valeur*, sous la forme d'un *tuple*. Voyons comment l'utiliser :

Code : Python Console

```
>>> fruits = {"pomme":21, "melon":3, "poire":31}
>>> for cle,valeur in fruits.items():
...     print("La clé {0} contient la valeur {1}.".format(cle,
valeur))
...
La clé melon contient la valeur 3.
La clé poire contient la valeur 31.
La clé pomme contient la valeur 21.
>>>
```

Il est parfois très pratique de parcourir un dictionnaire avec ses clés et les valeurs associées.

Entraînez-vous, il n'y a que cela de vrai. Pourquoi pas reprendre l'exercice du chapitre précédent, avec notre inventaire de fruits ? Sauf que le type de l'inventaire ne serait pas une liste mais un dictionnaire associant les noms des fruits aux quantités ?

Il nous reste une petite fonctionnalité supplémentaire à voir, et on en aura fini avec les dictionnaires.

Les dictionnaires et paramètres de fonction

Ça ne vous rappelle pas quelque chose ? J'espère bien que si, on a vu quelque chose similaire dans le chapitre précédent .

Si vous vous souvenez, on avait réussi à intercepter tous les paramètres de notre fonction... sauf les paramètres nommés.

Récupérer les paramètres nommés dans un dictionnaire

Il existe aussi une façon de capturer les paramètres nommés d'une fonction. Dans ce cas toutefois, ils sont placés dans un dictionnaire. Si par exemple vous appelez la fonction ainsi : `fonction(parametre='a')`, vous aurez, dans le dictionnaire capturant les paramètres nommés, une clé *'parametre'* liée à la valeur *'a'*. Voyez plutôt :

Code : Python Console

```
>>> def fonction_inconnue(**parametres_nommes):
...     """Fonction permettant de voir comment récupérer les
paramètres nommés
... dans un dictionnaire.
... """
...     print("J'ai reçu en paramètres nommés :
{0}.".format(parametres_nommes))
...
>>> fonction_inconnue() # aucun paramètre
J'ai reçu en paramètres nommés : {}
>>> fonction_inconnue(p=4, j=8)
J'ai reçu en paramètres nommés : {'p': 4, 'j': 8}
>>>
```

Pour capturer tous les paramètres nommés non précisés dans un dictionnaire, il faut mettre deux étoiles ** avant le nom du paramètre.

Si vous passez des paramètres non nommés à cette fonction, Python lèvera une exception.

Si bien que pour avoir une fonction qui accepte n'importe quel type de paramètres, nommés ou non, dans n'importe quel ordre, dans n'importe quelle quantité, il faut la déclarer ainsi :

Code : Python

```
def fonction_inconnue(*en_liste, **en_dictionnaire):
```

Tous les paramètres non nommés se retrouveront dans la variable *en_liste* et les paramètres nommés dans la variable *en_dictionnaire*.

Mais à quoi ça peut bien servir, d'avoir une fonction qui accepte n'importe quel paramètre ?

Pour l'instant à pas grand chose, mais ça viendra. Quand on abordera le chapitre sur les décorateurs, vous vous en souviendrez et vous pourrez vous féliciter de connaître cette fonctionnalité  .

Transformer un dictionnaire en paramètres nommés d'une fonction

Là encore, on peut faire exactement l'inverse : transformer un dictionnaire en paramètres nommés d'une fonction. Voyons un exemple tout simple :

Code : Python Console

```
>>> parametres = {"sep":" >> ", "end":" -\n"}
>>> print("Voici", "un", "exemple", "d'appel", **parametres)
Voici >> un >> exemple >> d'appel -
>>>
```

Les paramètres nommés sont transmis à la fonction par un dictionnaire. Pour indiquer à Python que le dictionnaire doit être transmis comme des paramètres nommés, on place deux étoiles avant son nom ** dans l'appel de la fonction.

Comme vous pouvez le voir, c'est comme si nous avions écrit :

Code : Python Console

```
>>> print("Voici", "un", "exemple", "d'appel", sep=" >> ", end="
-\n")
Voici >> un >> exemple >> d'appel -
>>>
```

Pour l'instant, vous devez trouver que c'est bien se compliquer la vie pour si peu. Nous verrons dans la suite de ce cours qu'il n'en est rien, en fait, même si nous n'utilisons pas cette fonctionnalité tous les jours  .

Ce chapitre a été moins long et, je pense, moins difficile que les deux derniers. Et ce n'est pas un hasard : vous commencez à vous habituer, du moins je l'espère, à la syntaxe de l'objet. De plus, les fonctionnalités que je décris ici commencent à vous sembler plus familières. Dans le prochain chapitre, nous parlerons des fichiers, un chapitre qui, lui aussi, devrait vous sembler on ne peut plus simple  .

Mais aussi, plus on avance, et plus je compte sur vous pour rechercher par vous-même, des informations sur des fonctionnalités, des objets, des méthodes. N'en soyez pas surpris : le but est de vous rendre autonome dans la recherche et, plus vous connaîtrez Python, plus ce sera simple vous verrez  .

Les fichiers

Poursuivons notre tour d'horizon des principaux objets. Nous allons voir dans ce chapitre les fichiers, comment les ouvrir, les lire, écrire dedans. Et nous verrons également comment sauvegarder nos objets dans des fichiers ... mais avant tout, voyons l'intérêt .

Avant de commencer

Nous allons beaucoup travailler sur des répertoires et des fichiers, autrement dit sur votre disque. Donc je vais vous donner quelques informations générales avant de commencer, pour que malgré vos différents systèmes et configurations, vous puissiez essayer les instructions que je vais vous montrer .

Mais d'abord pourquoi lire ou écrire dans des fichiers ?

Peut-être que vous ne voyez pas trop l'intérêt de savoir lire et écrire dans des fichiers, hors quelques applications de temps à autre. Mais souvenez-vous que, quand vous fermez votre programme, aucune de vos variables n'est sauvegardée. Or, les fichiers peuvent être, justement, un excellent moyen de garder les valeurs de certains objets pour pouvoir les récupérer quand vous rouvrirez votre programme. Par exemple, un petit jeu peut enregistrer les scores des joueurs.

Si dans notre TP *ZCasino* nous avions pu enregistrer, au moment de quitter le casino, la somme que nous avions en poche, nous aurions pu rejouer sans repartir de zéro.

Changer le répertoire de travail courant

Si vous souhaitez travailler dans l'interpréteur Python, et je vous y encourage, vous devrez changer le répertoire de travail courant. En effet, au lancement de l'interpréteur, le répertoire de travail courant est celui dans lequel se trouve l'exécutable de l'interpréteur. Sous Windows, c'est "C:\Python3X", le X étant différent en fonction de votre version de Python. Dans tous les cas, je vous invite à changer de répertoire de travail courant. Pour cela, vous devez utiliser une fonction du module *os*, qui s'appelle *chdir (Change Directory)*.

Code : Python Console

```
>>> import os
>>> os.chdir("C:/tests python")
>>>
```

Le répertoire doit exister. Modifier la chaîne passée en paramètre de *os.chdir* en fonction du dossier dans lequel vous souhaitez vous déplacer.

Je vous conseille, que vous soyez sous Windows ou non, d'utiliser le symbole / pour décrire un chemin.

Vous pouvez utiliser, en le doublant, l'antislash \\ mais si vous oubliez de le doubler, vous aurez des erreurs. Je vous conseille donc d'utiliser le slash /, cela marche très bien même sous Windows.

N.B. : quand vous lancez un programme Python directement, en double-cliquant dessus par exemple, le répertoire courant est celui où vous lancez le programme. Si vous avez un fichier *mon_programme.py* contenu sur le disque *C:*, le répertoire de travail courant quand vous lancerez le programme sera *C:*.

Chemins relatifs et absolus

Pour décrire l'arborescence d'un système, on a deux possibilités :

- Les chemins absolus
- Les chemins relatifs

Le chemin absolu

Quand on décrit une cible (un fichier ou un répertoire) sous la forme d'un chemin absolu, on décrit la suite des répertoires menant au fichier. Sous Windows, on partira du nom de volume (*C:*, *D:*...). Sous les systèmes Unix, ce sera plus vraissemblablement depuis /.

Par exemple, sous Windows, si on a un fichier nommé *fic.txt*, contenu dans un dossier *test*, lui-même présent sur le disque *C:*, notre chemin absolu menant à notre fichier sera *C:\test\fic.txt*.

Le chemin relatif

Quand on décrit la position d'un fichier grâce à un chemin relatif, cela veut dire que l'on tient compte du dossier dans lequel on se trouve actuellement. Ainsi, si on se trouve dans le dossier *C:\test* et que l'on souhaite accéder au fichier *fic.txt* contenu dans ce même dossier, le chemin relatif menant à ce fichier sera tout simplement *fic.txt*.

Maintenant, si on se trouve dans *C:*, notre chemin relatif sera *test\fic.txt*.

Quand on décrit un chemin relatif, on utilise parfois le symbole .. qui désigne le répertoire parent. Voici un nouvel exemple :

- C:
  - test
    - rep1
      - fic1.txt
    - rep2
      - fic2.txt
      - fic3.txt

C'est dans notre dossier *test* que tout se passe. Nous avons deux sous-répertoires nommés *rep1* et *rep2*. Dans *rep1*, nous avons un seul fichier : *fic1.txt*. Dans *rep2*, nous avons deux fichiers : *fic2.txt* et *fic3.txt*.

Si le répertoire de travail courant est *rep2* et que l'on souhaite accéder à *fic1.txt*, notre chemin relatif sera donc *..\rep1\fic1.txt*.

N.B. : j'utilise ici des anti-slash parce que l'exemple d'arborescence est un modèle Windows et que ce sont les séparateurs utilisés pour décrire une arborescence Windows. Mais dans votre code je vous conseille quand même d'utiliser un slash *(/)*.

Résumé

Les chemins absolus et relatifs sont donc deux moyens de décrire le chemin menant à des fichiers ou répertoires. Mais si le résultat est le même, le moyen utilisé n'est pas identique : quand on utilise un chemin absolu, on décrit toute l'arborescence menant au fichier, peu importe l'endroit où on se trouve. Un chemin absolu permet d'accéder à un endroit dans le disque peu importe le répertoire de travail courant. L'inconvénient de cette méthode, c'est qu'on doit savoir en général où se trouvent les fichiers qui nous intéressent sur le disque.

Le chemin relatif décrit l'arborescence en prenant comme point d'origine non pas la racine, ou le périphérique sur lequel est stocké la cible, mais le répertoire dans lequel on se trouve. Ça a certains avantages quand on code un projet, on est pas obligé de savoir où le projet est stocké pour construire plusieurs répertoires. Mais ce n'est pas forcément la meilleure solution en toute circonstance.

Comme je l'ai dit, quand on lance l'interpréteur Python, on a bel et bien un répertoire de travail courant. Vous pouvez l'afficher grâce à la fonction *os.getcwd()* (CWD = "Current Working Directory").

Cela devrait vous suffir donc. Pour les démonstrations qui vont suivre, placez-vous à l'aide de *os.chdir* dans un répertoire de test créé pour l'occasion.

Lecture et écriture dans un fichier

Nous allons commencer à lire avant d'écrire dans un fichier. Pour l'exemple donc, je vous invite à créer un fichier dans le répertoire de travail courant que vous avez choisi. Je suis en manque flagrant d'inspiration, je vais l'appeler *fichier.txt* et je vais écrire dedans, à l'aide d'un éditeur sans mise en forme (tel que le bloc-notes Windows) : *« C'est le contenu du fichier. Spectaculaire non ? »*

Ouverture du fichier

D'abord, il nous faut ouvrir le fichier avec Python. On utilise pour ce faire la fonction *open*, disponible sans avoir besoin de rien importer. Elle prend en paramètres :

- le chemin (absolu ou relatif) menant au fichier à ouvrir ;
- le mode d'ouverture.

Le mode est donné sous la forme d'une chaîne de caractères. Voici les principaux modes :

Mode	Explications
'r'	Ouverture en lecture *(Read)*.
'w'	Ouverture en écriture *(Write)*. Le contenu du fichier est écrasé. Si le fichier n'existe pas, il est créé.
'a'	Ouverture en écriture en mode ajout *(Append)*. On écrit à la fin du fichier sans écraser l'ancien contenu du fichier. Si le fichier n'existe pas, il est créé.

On peut ajouter à tous ces modes le signe *b* pour ouvrir le fichier en mode binaire. Nous verrons plus loin l'utilité, c'est un mode un peu particulier.

Ici nous souhaitons lire le fichier. Nous allons donc utilisé le mode *'r'*.

Code : Python Console

```
>>> mon_fichier = open("fichier.txt", "r")
>>> mon_fichier
<_io.TextIOWrapper name='fichier.txt' encoding='cp1252'>
>>> type(mon_fichier)
<class '_io.TextIOWrapper'>
>>>
```

L'encodage précisé quand on affiche le fichier dans l'interpréteur peut être très différent en fonction de votre système. Ici je suis dans l'interpréteur Python dans Windows et l'encodage choisi est donc un encodage Windows propre à la console. Ne soyez pas surpris s'il est différent chez vous.

La fonction *open* crée donc un fichier. Elle retourne un objet issue de la classe *TextIoWrapper*. Par la suite, nous allons utiliser des méthodes de cette classe pour interragir avec le fichier.

Le type de l'objet doit vous surprendre quelque peu. Ça aurait très bien pu être un type *file* après tout. En fait, *open* permet d'ouvrir un fichier, mais *TextIoWrapper* est utilisé dans d'autres circonstances, pour afficher du texte à l'écran par exemple. Bon, ça ne nous concerne pas trop ici, je ne vais pas m'y attarder  .

Fermer le fichier

N'oubliez pas de fermer un fichier après l'avoir ouvert. Si d'autres applications souhaitent accéder à ce fichier, ou d'autres morceaux de votre propre code, ils ne pourront pas car le fichier sera déjà ouvert. C'est surtout vrai en écriture, mais prenez de bonnes habitudes  . La méthode à utiliser est *close* :

Code : Python Console

```
>>> mon_fichier.close()
>>>
```

Lire l'intégralité du fichier

Pour ce faire, on utilise la méthode *read* de la classe *TextIoWrapper*. Elle retourne l'intégralité du fichier :

Code : Python Console

```
>>> mon_fichier = open("fichier.txt", "r")
>>> contenu = mon_fichier.read()
>>> print(contenu)
C'est le contenu du fichier. Spectaculaire non ?
>>> mon_fichier.close()
>>>
```

Quoi de plus simple ? La méthode *read* retourne tout le fichier que l'on capture dans une chaîne de caractères. Notre fichier ne contient pas de saut de ligne, mais si c'était le cas, vous auriez dans votre variable *contenu* les signes *\n* traduisant un saut de ligne.

Maintenant que vous avez une chaîne, vous pouvez naturellement tout faire : la convertir, tout entière ou en partie, si c'est nécessaire, *split* la chaîne pour parcourir chaque ligne et les traiter... bref, tout est possible .

Écriture dans un fichier

Bien entendu, il nous faut ouvrir le fichier avant tout. Vous pouvez utiliser le mode *w* ou le mode *a*. Le premier écrase le contenu éventuel du fichier, alors que le second ajoute ce que l'on écrit à la fin du fichier. À vous de voir en fonction de vos besoins .
Dans tous les cas, ces deux modes créent le fichier s'il n'existe pas.

Écrire une chaîne

Pour écrire dans un fichier, on utilise la méthode *write* en lui passant en paramètre la chaîne à écrire dans le fichier. Elle retourne le nombre de caractères qui ont été écrits. On n'est naturellement pas obligé de récupérer cette valeur, sauf si on en a besoin.

Code : Python Console

```
>>> mon_fichier = open("fichier.txt", "w") # argh j'ai tout écrasé !
>>> mon_fichier.write("Premier test d'écriture dans un fichier via
Python")
50
>>> mon_fichier.close()
>>>
```

Vous pouvez vérifier que votre fichier contient bien le texte qu'on y a écrit.

Écrire d'autres types de données

La méthode *write* n'accepte en paramètre que des chaînes de caractères. Si vous voulez écrire dans votre fichier des nombres, des scores par exemple, il vous faudra les convertir en chaîne avant de les écrire, et les convertir en entier après les avoir lu.

Le module *os* contient beaucoup de fonctions intéressantes pour créer et supprimer des fichiers et des répertoires. Je vous laisse regarder l'aide si vous êtes intéressé .

Le mot-clé with

Ne désespérez pas, il ne nous reste plus autant de mots-clés à découvrir... mais quelques-uns tout de même. Et même certains dont je ne parlerai pas...

On n'est jamais à l'abri d'une erreur. Surtout quand on manipule des fichiers. Il peut se produire des erreurs quand on lit, quand on écrit... et si l'on n'y prend garde, le fichier restera ouvert.

Comme je vous l'ai dit, c'est plutôt gênant, et ça peut même être grave. Si votre programme souhaite de nouveau utiliser ce fichier, il ne pourra pas forcément y accéder, puisqu'il a déjà été ouvert.

Il existe un mot-clé qui permet d'éviter cette situation : *with*. Voici sa syntaxe :

Code : Python

```
with open(mon_fichier, mode_ouverture) as variable:
    # opérations sur le fichier
```

On trouve dans l'ordre :

- Le mot-clé *with*, prélude au bloc dans lequel on va manipuler notre fichier. On peut trouver *with* dans la manipulation d'autres objets, mais nous ne le verrons pas ici
- Notre objet. Ici, on appelle *open* qui va retourner un objet *TextIOWraper* (notre fichier)
- Le mot-clé *as* que nous avons déjà vu dans le mécanisme d'importation et dans les exceptions. Il signifie toujours la même chose : "en tant que"
- Notre variable qui contiendra notre objet. Si la variable n'existe pas, Python la créée.

Un exemple ?

Code : Python Console

```
>>> with open('fichier.txt', 'r') as mon_fichier:
...     texte = mon_fichier.read()
...
>>>
```

Cela ne veut pas dire que le bloc d'instructions ne lèvera aucune exception.

Cela signifie simplement que, si une exception se produit, le fichier sera tout de même fermé à la fin du bloc.

Vous pouvez appeler *mon_fichier.closed* pour le vérifier. Si le fichier est fermé, *mon_fichier.closed* vaudra *True*.

Il est inutile par conséquent de fermer le fichier à la fin du bloc *with*. Python va le faire tout seul, qu'une exception soit levée ou non. Je vous encourage à utiliser cette syntaxe, elle est plus sûre et plus facile à comprendre.

Allez ! Direction le module *pickle*, dans lequel nous allons apprendre à sauvegarder dans des fichiers nos objets.

Enregistrer des objets dans des fichiers

Dans beaucoup de langages de haut niveau, on peut enregistrer ses objets dans un fichier. Python ne fait pas exception. Grâce au module *pickle* que nous allons découvrir, on peut enregistrer n'importe quel objet et le récupérer par la suite, au prochain lancement du programme, par exemple. En outre, le fichier résultant pourra être lu depuis n'importe quel système d'exploitation (supportant Python, naturellement).

Enregistrer un objet dans un fichier

Il nous faut naturellement d'abord importer le module *pickle*.

Code : Python Console

```
>>> import pickle
>>>
```

On va ensuite utiliser deux classes incluses dans ce module : la classe *Pickler* et la classe *Unpickler*.

C'est la première qui nous intéresse dans cette partie.

Pour créer notre objet *Pickler*, on va l'appeler en passant en paramètre le fichier dans lequel on va enregistrer notre objet.

Code : Python Console

```
>>> with open('donnees', 'wb') as fichier:
...     mon_pickler = pickle.Pickler(fichier)
...     # enregistrement ...
...
>>>
```

Quand on va enregistrer nos objets, ce sera dans le fichier *donnees*. Je ne lui ai pas donné d'extension, vous pouvez le faire. Mais évitez de préciser une extension qui est utilisée par un programme.

Notez le mode d'ouverture : on ouvre le fichier *donnees* en mode d'écriture binaire. Il suffit de rajouter derrière le nom du mode la lettre *b* pour indiquer un mode binaire.

Le fichier que Python va écrire ne sera pas très lisible si vous essayez de l'ouvrir, et ce n'est pas le but  .

Bon. Maintenant que notre pickler est créé, on va enregistrer un ou plusieurs objets dans notre fichier. Là, c'est à vous de voir comment vous voulez vous organiser, ça dépend aussi beaucoup du projet. Moi j'ai pris l'habitude de n'enregistrer qu'un objet par fichier, mais il n'y a aucune obligation.

On utilise la méthode *dump* du pickler pour enregistrer notre objet. Son utilisation est des plus simples :

Code : Python Console

```
>>> score = {
...   "joueur 1":    5,
...   "joueur 2":   35,
...   "joueur 3":   20,
...   "joueur 4":    2,
>>> }
>>> with open('donnees', 'wb') as fichier:
...     mon_pickler = pickle.Pickler(fichier)
...     mon_pickler.dump(score)
...
>>>
```

Après l'exécution de ce code, vous avez dans votre dossier de test un fichier *donnees* qui contient... eh bien, notre dictionnaire contenant les scores de nos quatre joueurs. Si vous voulez enregistrer plusieurs objets, appelez de nouveau la méthode *dump* avec les objets à enregistrer. Ils seront ajoutés dans le fichier dans l'ordre où vous les enregistrez.

Récupérer nos objets enregistrés

Nous allons utiliser une autre classe définie dans notre module *pickle*. Cette fois, assez logiquement, c'est la classe *Unpickler*.

Commençons par créer notre objet. On lui passe lors de la création notre fichier dans lequel on va lire nos objets. Puisqu'on va lire, on change de mode, on repasse en mode *r*, et même *rb* puisque le fichier est binaire.

Code : Python Console

```
>>> with open('donnees', 'rb') as fichier:
...     mon_depickler = pickle.Unpickler(fichier)
...     # lecture des objets contenus dans le fichier...
```

```
...
>>>
```

Pour lire l'objet dans notre fichier, il faut appeler la méthode *load* de notre depickler. Elle retourne le premier objet qui a été lu (s'il y en a plusieurs, il faut l'appeler plusieurs fois).

Code : Python Console

```
>>> with open('donnees', 'rb') as fichier:
...     mon_depickler = pickle.Unpickler(fichier)
...     score_recupere = mon_depickler.load()
...
>>>
```

Et après cet appel, si le fichier a pu être lu, dans votre variable *score_recupere*, vous récupérez votre dictionnaire contenant les scores. Là, c'est peut-être peu spectaculaire, mais quand vous utilisez ce module pour sauvegarder des objets qui doivent être conservés alors que votre programme n'est pas lancé, c'est franchement très pratique .
Et voilà . Nous approchons à grands pas de la fin de cette partie. Dans le prochain chapitre, nous ne découvrirons aucune nouvelle classe, mais nous allons nous intéresser d'un peu plus près au mécanisme des références, un mot que j'ai déjà utilisé dans ce cours mais jamais bien expliqué. Vous ne découvrirez aucune nouvelle classe, et vous pouvez continuer d'apprendre le Python sans lire ce chapitre... mais je vous conseille tout de même d'y passer .

Portée des variables et références

Dans ce chapitre, je vais m'attarder sur la portée des variables et sur les références. Je ne vais pas vous faire une visite guidée de la mémoire de votre ordinateur (Python est assez haut niveau pour, justement, ne pas avoir à descendre aussi bas), je vais simplement souligner quelques cas intéressants que vous pourriez rencontrer dans vos programmes.

Ce chapitre n'est pas indispensable, mais je ne l'écris naturellement pas pour le plaisir : vous pouvez très bien continuer à apprendre le Python sans connaître précisément comment Python joue avec les références, mais il peut être utile de le savoir.

N'hésitez pas à relire ce chapitre si vous avez un peu de mal, les concepts présentés ne sont pas évidents .

La portée des variables

En Python, comme dans la plupart des langages, on trouve des règles qui définissent la portée des variables. La portée utilisée dans ce sens c'est « quand et comment les variables sont accessibles ». Quand vous définissez une fonction, quelles variables sont utilisables dans son corps ? Uniquement les paramètres ? Est-ce qu'on peut créer dans notre corps de fonction des variables utilisables en dehors ? Si vous ne vous êtes jamais posé ces questions, c'est normal . Mais je vais tout de même y répondre, car elles ne sont pas dénuées d'intérêt .

Dans nos fonctions, quelles variables sont accessibles ?

On ne change pas une équipe qui gagne : passons aux exemples dès à présent.

Code : Python Console

```
>>> a = 5
>>> def print_a():
...     """Fonction chargée d'afficher la variable a.
... Cette variable a n'est pas passée en paramètre de la fonction.
... On suppose qu'elle a été créée en dehors de la fonction, on
veut voir
... si elle est accessible depuis le corps de la fonction.
... """
...     print("La variable a = {0}.".format(a))
...
>>> print_a()
La variable a = 5.
>>> a = 8
>>> print_a()
La variable a = 8.
>>>
```

Surprise ! Ou peut-être pas…

La variable *a* n'est pas passée en paramètre de la fonction *print_a*. Et pourtant, Python la trouve, tant qu'elle a été définie avant l'appel de la fonction.

C'est là qu'interviennent les différents espaces.

L'espace local

Dans votre fonction, quand vous faites référence à une variable *a*, Python vérifie dans l'**espace local** de la fonction. Cet espace contient les paramètres qui sont passés à la fonction, et les variables définies dans son corps. Python apprend ainsi que la variable *a* n'existe pas dans l'espace local de la fonction. Dans ce cas, il va regarder dans l'espace local dans lequel la fonction a été appelée. Et là, il trouve bien la variable *a* et peut donc l'afficher.

D'une façon générale, je vous conseille d'éviter d'appeler des variables qui ne sont pas dans l'espace local, sauf si c'est nécessaire. Ce n'est pas très clair à la lecture ; dans l'absolu, préférez faire des variables globales, ça reste plus propre (nous verrons ça plus bas). Pour l'instant, on ne s'intéresse qu'aux mécanismes, on cherche juste à savoir quelles variables sont accessibles depuis un corps de fonction, et de quelle façon.

La portée de nos variables

Voyons quelques cas concrets. Je vais les expliquer au fur et à mesure, ne vous en faites pas .

Qu'advient-il des variables définies dans un corps de fonction ?

Voyons un nouvel exemple :

Code : Python

```
def set_var(nouvelle_valeur):
    """Fonction nous permettant de tester la portée des variables
définies dans notre corps de fonction.
"""
    # On essaye d'afficher la variable var, si elle existe
    try:
        print("Avant l'affectation, notre variable var vaut
{0}.".format(var))
    except NameError:
        print("La variable var n'existe pas encore.")
    var = nouvelle_valeur
    print("Après l'affectation, notre variable var vaut
{0}.".format(var))
```

Et maintenant, utilisons notre fonction :

Code : Python Console

```
>>> set_var(5)
La variable var n'existe pas encore.
Après l'affectation, notre variable var vaut 5.
>>> var
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'var' is not defined
>>>
```

Je sens que quelques explications s'imposent :

- Lors de notre appel à *set_var*, notre variable *var* n'a pu être trouvée par Python : c'est normal, nous ne l'avons pas encore définie, ni dans notre corps de fonction, ni dans le corps de notre programme. Python affecte la valeur *5* à la variable *var*, l'affiche et s'arrête.
- Au sortir de la fonction, on essaye d'afficher la variable *var*… mais Python ne la trouve pas ! En effet : elle a été définie dans le corps de la fonction (donc dans son espace local) et, à la fin de l'exécution de la fonction, l'espace est détruit… donc la variable *var*, définie dans le corps de la fonction, n'existe que dans ce corps et est détruite ensuite.

Python a une règle d'accès spécifique aux variables extérieures à l'espace local : on peut les lire, mais pas les modifier. C'est pourquoi, dans notre fonction *print_a*, on arrivait à afficher une variable qui n'était pas comprise dans l'espace local de la fonction. En revanche, on ne peut modifier la valeur d'une variable extérieure à l'espace local, par affectation du moins. Si dans votre corps de fonction vous faites `var = nouvelle_valeur` , vous n'allez en aucun cas modifier une variable extérieure au corps.

En fait, quand Python trouve une instruction d'affectation, comme `var = nouvelle_valeur` , il va changer la valeur de la variable dans l'espace local de la fonction. Et rappelez-vous que cet espace local est détruit après l'appel à la fonction.

Pour résumer, et c'est ce qu'il faut retenir, une fonction ne peut modifier, par affectation, la valeur d'une variable extérieure à son espace local.

Ça paraît plutôt stupide au premier abord… mais pas d'impatience. Je vais relativiser ça assez rapidement .

Une fonction modifiant des objets

J'espère que vous vous en souvenez, tout est objet en Python. Quand vous passez des paramètres à votre fonction, ce sont des objets qui sont transmis. Et pas les valeurs des objets, mais bien les objets eux-mêmes, ceci est très important.

Bon. On ne peut affecter une nouvelle valeur à un paramètre dans le corps de la fonction. Je ne reviens pas là-dessus. En revanche, on pourrait essayer d'appeler une méthode de l'objet qui le modifie… Voyons cela :

Code : Python Console

```
>>> def ajouter(liste, valeur_a_ajouter):
...     """Cette fonction ajoute la valeur que l'on veut ajouter, à
la fin de liste.
... """
...     liste.append(valeur_a_ajouter)
...
>>> ma_liste=['a', 'e', 'i']
>>> ajouter(ma_liste, 'o')
>>> ma_liste
['a', 'e', 'i', 'o']
>>>
```

Ça marche ! On passe en paramètres notre objet de type *list* avec la valeur à ajouter. Et la fonction appelle la méthode *append* de l'objet. Cette fois, au sortir de la fonction, notre objet a bel et bien été modifié.

Je vois pas pourquoi. Tu as dit qu'une fonction ne pouvait pas affecter de nouvelles valeurs aux paramètres ?

Absolument. Mais c'est ça la petite subtilité dans l'histoire : on ne change pas du tout la valeur du paramètre, on appelle juste une méthode de l'objet. Et ça change tout. Si vous vous embrouillez, retenez que dans le corps de fonction, si vous faites *parametre = nouvelle_valeur*, le paramètre ne sera modifié que dans le corps de la fonction. Alors que si vous faites *parametre.methode_pour_modifier(...)*, l'objet derrière le paramètre sera bel et bien modifié.

On peut aussi modifier les attributs d'un objet, par exemple, changer une case de la liste ou d'un dictionnaire, ces changements aussi seront effectifs au-delà de l'appel de la fonction.

Et les références, dans tout ça ?

J'ai parlé des références, je dis y consacrer un chapitre, et jusqu'ici… mais c'est maintenant qu'on en parle .

Je vais schématiser volontairement : les variables que nous utilisons depuis le début de ce cours cachent en fait des références vers des objets.

Concrètement, j'ai présenté les variables comme ceci : un nom identifiant pointant vers une valeur. Par exemple, notre variable nommée *a* possède une valeur (0, pourquoi pas).

En fait, une variable est un nom identifiant, pointant vers une référence d'un objet. La référence, c'est un peu sa position en mémoire. Ça reste plus haut niveau que les pointeurs en C par exemple, ce n'est pas vraiment la mémoire de votre ordinateur. Et on ne manipule pas ces références directement.

Cela signifie que deux variables peuvent pointer sur le même objet.

Bah… bien sûr, rien n'empêche de faire deux variables avec la même valeur.

Non non, je ne parle pas de valeurs ici , mais d'objets. Voyons un exemple, vous allez comprendre :

Code : Python Console

```
>>> ma_liste1 = [1, 2, 3]
>>> ma_liste2 = ma_liste1
>>> ma_liste2.append(4)
>>> print(ma_liste2)
[1, 2, 3, 4]
>>> print(ma_liste1)
[1, 2, 3, 4]
>>>
```

Nous créons une liste dans la variable *ma_liste1*. À la ligne 2, nous affectons *ma_liste1* à la variable *ma_liste2*. On pourrait croire que *ma_liste2* est une copie de *ma_liste1*. Toutefois, quand on ajoute *4* à *ma_liste2*, *ma_liste1* est aussi modifiée.

On dit que *ma_liste1* et *ma_liste2* contiennent une référence vers le même objet : si on modifie l'objet depuis une des deux variables, le changement sera visible depuis les deux variables.

Euh… j'essaye de faire la même chose avec des variables contenant des entiers… ça ne marche pas.

C'est normal. Les entiers, les flottants, les chaînes de caractères, n'ont aucune méthode travaillant sur l'objet lui-même. Les chaînes de caractères, comme nous l'avons vu, ne modifient pas l'objet appelant mais retournent un nouvel objet modifié. Et comme nous venons de le voir, le processus d'affectation n'est pas du tout identique à un appel de méthode.

Et si je veux modifier une liste sans toucher à l'autre ?

Eh bien c'est impossible, vu comment nous avons défini nos listes. Les deux variables pointent sur le même objet par jeu de références et donc, inévitablement, si vous modifiez l'objet, vous allez voir le changement depuis les deux variables. Toutefois, il existe un moyen pour créer un nouvel objet depuis un autre :

Code : Python Console

```
>>> ma_liste1 = [1, 2, 3]
>>> ma_liste2 = list(ma_liste1) # ça revient à copier le contenu de
ma_liste1
>>> ma_liste2.append(4)
>>> print(ma_liste2)
[1, 2, 3, 4]
>>> print(ma_liste1)
[1, 2, 3]
>>>
```

À la ligne 2, nous avons demandé à Python de créer un nouvel objet en le fondant sur *ma_liste1*. Du coup, les deux variables ne contiennent plus la même référence : elles modifient des objets différents. Vous pouvez utiliser la plupart des constructeurs (c'est le nom qu'on donne à *list* pour créer une liste par exemple) dans ce but. Pour des dictionnaires, utilisez le constructeur *dict* en lui passant en paramètre un dictionnaire déjà construit et vous aurez en retour un dictionnaire, semblable à celui passé en paramètre, mais seulement semblable par le contenu. En fait, il s'agit d'une copie de l'objet, ni plus ni moins.

Pour approcher de plus près les références, vous avez la fonction *id* qui prend en paramètre un objet. Elle retourne la position de l'objet dans la mémoire Python sous la forme d'un entier (plutôt grand). Je vous invite à faire quelques tests en passant en paramètre de cette fonction plusieurs objets. Sachez au passage que *is* compare les ID des objets de part et d'autre et c'est pour cette raison que je vous ais mis en garde quant à son utilisation.

Code : Python Console

```
>>> ma_liste1 = [1, 2]
```

```
>>> ma_liste2 = [1, 2]
>>> ma_liste1==ma_liste2 # on compare le contenu des listes
True
>>> ma_liste1 is ma_liste2 # on compare leur référence
False
>>>
```

Je ne peux que vous encourager à faire des tests avec différents objets. Un petit tour du côté des variables globales ?

Les variables globales

Il existe un moyen de modifier, dans une fonction, des variables extérieures à celle-ci. On utilise pour cela des variables globales.

Cette distinction entre variables locales et variables globales se retrouve dans d'autres langages et on encourage souvent d'éviter de trop les utiliser. Elles peuvent avoir leurs utilités, toutefois, puisque le mécanisme existe. D'un point de vue strictement personnel, tant que c'est possible je ne travaille qu'avec des variables locales (comme nous l'avons fait depuis le début de ce cours), mais il m'arrive de faire appel à des variables globales quand c'est nécessaire ou bien plus pratique. Mais ne tombez pas dans l'extrême non plus, ni dans un sens ni dans l'autre .

Le principe des variables globales

On ne peut faire plus simple. On déclare dans le corps de notre programme, donc en dehors de tout corps de fonction, une variable, tout ce qu'il y a de plus normal. Dans le corps d'une fonction qui doit modifier cette variable (changer sa valeur par affectation), on déclare à Python que la variable qui doit être utilisée dans ce corps est globale.

Python va regarder dans les différents espaces : celui de la fonction, celui dans lequel la fonction a été appelée… ainsi de suite jusqu'à mettre la main sur notre variable. S'il la trouve, il va nous donner le plein accès à cette variable dans le corps de la fonction.

Cela signifie que nous pouvons y accéder en lecture (comme c'est le cas sans avoir besoin de la définir comme variable globale) mais aussi en écriture. Une fonction peut donc ainsi changer la valeur d'une variable directement .

Mais assez de théorie, voyons un exemple.

Utiliser concrètement les variables globales

Pour déclarer à Python, dans le corps d'une fonction, que la variable qui sera utilisée doit être considérée comme globale, on utilise le mot-clé *global*. On le place généralement après la définition de la fonction, juste en-dessous de la *docstring*, ça permet de vite retrouver les variables globales sans parcourir tout le code (c'est une simple convention). On précise derrière ce mot-clé le nom de la variable à considérer comme globale :

Code : Python Console

```
>>> i = 4 # une variable, nommée i, contenant un entier
>>> def inc_i():
...     """Fonction chargée d'incrémenter i de 1"""
...     global i # Python recherche i en dehors de l'espace local
de la fonction
...     i += 1
...
>>> i
4
>>> inc_i()
>>> i
5
>>>
```

Si vous ne précisez pas à Python que *i* doit être considérée comme globale, vous ne pourrez pas modifier réellement sa valeur, comme nous l'avons vu plus haut. En précisant *global i*, Python permet l'accès en lecture et en écriture à cette variable, ce qui signifie que vous pouvez changer sa valeur par affectation.

J'utilise ce mécanisme quand je travaille sur plusieurs classes et fonctions qui doivent s'échanger des informations d'état par exemple. Il existe d'autres moyens, mais vous connaissez celui-ci et tant que vous maîtrisez bien votre code, il n'est pas plus mauvais qu'un autre .

Je vous dispense de QCM sur ce chapitre, vous avez de la chance .

J'espère avoir été assez clair dans mes explications. Ne restez pas trop bloqué sur le concept des références, des espaces, des variables locales et globales si vous n'en discernez pas la logique. Le déclic se fera peut-être plus tard, quand vous serez directement confronté à la difficulté, si cela se produit. En attendant, je crois qu'un petit TP s'impose, qu'en pensez-vous ?

TP 2 : un bon vieux petit pendu

C'est le moment de mettre en pratique ce que vous avez appris. Vous n'aurez pas besoin de tout, bien entendu, mais je vais essayer de vous faire travailler le maximum de choses .

Nous allons donc faire un jeu de pendu plutôt classique. Ce n'est pas bien original, mais mes autres idées n'étaient pas très adaptées. Mais on va compliquer un peu l'exercice, vous allez voir .

Votre mission

Nous y voilà. Je vais vous préciser un peu la mission sans quoi, on va avoir du mal à s'entendre sur la correction .

Un jeu du pendu

Le premier point de la mission est de réaliser un jeu du pendu. Je rappelle brièvement la règle, au cas où : l'ordinateur choisit un mot au hasard dans une liste, un mot de huit lettres maximum. Le joueur tente de trouver les lettres composant le mot. À chaque coup, il entre une lettre. Si la lettre est dans le mot, l'ordinateur affiche le mot avec les lettres déjà trouvées. Celles qui ne le sont pas encore sont remplacées par des étoiles (*). Le joueur a 8 chances. Au-delà, il a perdu.

On va compliquer un peu la règle en faisant en sorte qu'au début de la partie, le joueur doive entrer son nom. Cela permettra au programme d'enregistrer son score.

Le score du joueur sera simple à calculer : on prend le score courant (0 si le joueur n'a aucun score déjà enregistré) et à chaque partie, on lui ajoute le nombre de coups restants comme points de partie. Si par exemple il me reste trois chances au moment où je trouve le mot, je gagne trois points.

Par la suite, vous pourrez vous amuser à faire un décompte du score plus poussé, pour l'instant ça suffira bien.

Le côté technique du problème

Le jeu du pendu en lui-même, vous ne devriez avoir aucun problème pour le mettre en place. Rappelez-vous que le joueur ne doit entrer qu'une seule lettre à la fois et que le programme doit bien vérifier que c'est le cas avant de continuer. Nous allons découper notre programme en trois fichiers :

- Le fichier *donnees.py* qui contiendra les variables nécessaires à notre application (la liste des mots, le nombre de chances autorisées…).
- Le fichier *fonctions.py* qui contiendra les fonctions utiles à notre application. Là, je ne vous fais aucune liste claire, je vous conseille de bien y réfléchir, avec une feuille et un stylo si ça vous aide (Quelles sont les actions de mon programme ? Que puis-je mettre dans des fonctions ?).
- Enfin, notre fichier *pendu.py* qui contiendra notre jeu du pendu.

Gérer les scores

Vous avez, j'espère, une petite idée de comment faire cela… mais je vais quand même clarifier : on va enregistrer dans un fichier de données, que l'on va appeler *scores* (sans aucune extension) les scores du jeu. Ces scores seront sous la forme d'un dictionnaire : en clé, nous aurons les noms des joueurs, et en valeur les scores sous la forme d'entiers.

Il faut gérer les cas suivants :

- Le fichier n'existe pas. Là, on créée un dictionnaire vide, aucun score n'a été trouvé.
- Le joueur n'est pas dans le dictionnaire. Dans ce cas, on l'ajoute avec un score de *0*.

À vous de jouer

Vous avez l'essentiel. Peut-être pas tout ce dont vous avez besoin, cela dépend de comment vous vous organisez, mais le but est aussi de chercher ce dont vous avez besoin. Encore une fois, c'est un exercice pratique, ne sautez pas à la correction, ça ne vous apprendra pas grand chose .

Bonne chance !

Correction proposée

Voici la correction que je vous propose. J'espère que vous êtes arrivé à un résultat satisfaisant, même si vous n'avez pas forcément réussi à tout faire. Si votre jeu marche, c'est parfait !

Secret (cliquez pour afficher)

Voici le code des trois fichiers.

donnees.py

Code : Python

```
"""Ce fichier définit quelques données, sous la forme de
variables,
utiles au programme pendu.

"""

# Nombre de coups par partie
nb_coups = 8

# Nom du fichier stockant les scores
nom_fichier_scores = "scores"

# Liste des mots du pendu
liste_mots = [
    "armoire",
    "boucle",
    "buisson",
    "bureau",
    "chaise",
    "carton",
    "couteau",
    "fichier",
    "garage",
    "glace",
    "journal",
    "kiwi",
    "lampe",
    "liste",
    "montagne",
    "remise",
    "sandale",
    "taxi",
    "vampire",
    "volant",
]
```

fonctions.py

Code : Python

```
"""Ce fichier définit des fonctions utiles pour le programme pendu.

On utilise les données du programme contenues dans donnees.py.

"""

import os
import pickle
from random import choice
```

```
from donnees import *

# Gestion des scores

def recup_scores():
    """Cette fonction récupère les scores enregistrés si le fichier existe.
    Dans tous les cas, on retourne un dictionnaire, soit l'objet dépicklé,
    soit un dictionnaire vide.

    On s'appuie sur nom_fichier_scores définit dans donnees.py.

    """
    if os.path.exists(nom_fichier_scores): # le fichier existe
        # On le récupère
        fichier_scores = open(nom_fichier_scores, "rb")
        mon_depickler = pickle.Unpickler(fichier_scores)
        scores = mon_depickler.load()
        fichier_scores.close()
    else: # le fichier n'existe pas
        scores = {}
    return scores

def enregistrer_scores(scores):
    """Cette fonction se charge d'enregistrer les scores dans le fichier
    nom_fichier_scores . Elle reçoit en paramètre le dictionnaire des scores
    à enregistrer.

    """
    fichier_scores = open(nom_fichier_scores, "wb") # on écrase les anciens
    mon_pickler = pickle.Pickler(fichier_scores)
    mon_pickler.dump(scores)
    fichier_scores.close()


# Fonction gérant les entrées d'utilisateur

def recup_nom_utilisateur():
    """Fonction chargée de récupérer le nom de l'utilisateur.
    Le nom de l'utilisateur doit être composé de 4 caractères minimum,
    chiffres et lettres exclusivement.

    Si ce nom n'est pas valide, on appelle récursivement la fonction
    pour en obtenir un nouveau.

    """
    nom_utilisateur = input("Entrez votre nom: ")
    # On met la première lettre en majuscule et les autres en minuscule
    nom_utilisateur = nom_utilisateur.capitalize()
    if not nom_utilisateur.isalnum() or len(nom_utilisateur)<4:
        print("Ce nom est invalide.")
        # On appelle de nouveau la fonction pour avoir un autre nom
        return recup_nom_utilisateur()
    else:
        return nom_utilisateur

def recup_lettre():
    """Cette fonction est chargée de récupérer une lettre entrée par
    l'utilisateur. Si la chaîne récupérée n'est pas une lettre,
    on appelle récursivement la fonction jusqu'à obtenir une lettre.

    """
    lettre = input("Entrez une lettre: ")
    lettre = lettre.lower()
    if len(lettre)>1 or not lettre.isalpha():
        print("Vous n'avez pas entré une lettre valide.")
        return recup_lettre()
    else:
        return lettre
```

```
    # Fonctions du jeu de pendu

    def choisir_mot():
        """Cette fonction retourne le mot choisi dans la liste des mots
        liste_mots .

        On utilise la fonction choice du module random (voir l'aide)

        """
        return choice(liste_mots)

    def recup_mot_masque(mot_complet, lettres_trouvees):
        """Cette fonction retourne un mot masqué tout ou en partie, en fonction
        - du mot d'origine (type str)
        - des lettres déjà trouvées (type list)

        On retourne le mot d'origine avec des * remplaçant les lettres que l'on
        n'a pas encore trouvées.

        """
        mot_masque = ""
        for lettre in mot_complet:
            if lettre in lettres_trouvees:
                mot_masque += lettre
            else:
                mot_masque += "*"
        return mot_masque
```

pendu.py

Code : Python

```
    """Ce fichier contient le jeu du pendu.

    Il s'appuie sur les fichiers :
    - donnees.py
    - fonctions.py
    """

    from donnees import *
    from fonctions import *

    # On récupère les scores de la partie
    scores = recup_scores()

    # On récupère un nom d'utilisateur
    utilisateur = recup_nom_utilisateur()

    # Si l'utilisateur n'a pas encore de score, on l'ajoute
    if utilisateur not in scores.keys():
        scores[utilisateur] = 0 # 0 point pour commencer

    # Notre variable pour savoir quand arrêter la partie
    continuer_partie = 'o'

    while continuer_partie != 'n':
        print("Joueur {0}: {1} point(s)".format(utilisateur,
    scores[utilisateur]))
        mot_a_trouver = choisir_mot()
        lettres_trouvees = []
        mot_trouve = recup_mot_masque(mot_a_trouver, lettres_trouvees)
        nb_chances = nb_coups
        while mot_a_trouver!=mot_trouve and nb_chances>0:
            print("Mot à trouver {0} (encore {1}
    chances)".format(mot_trouve, nb_chances))
```

```
            lettre = recup_lettre()
            if lettre in lettres_trouvees: # la lettre a déjà été
entrée
                print("Vous avez déjà entré cette lettre.")
            elif lettre in mot_a_trouver: # la lettre est dans le mot
à trouver
                lettres_trouvees.append(lettre)
                print("Bien joué.")
            else:
                nb_chances -= 1
                print("... non, cette lettre ne se trouve pas dans le
mot ...")
            mot_trouve = recup_mot_masque(mot_a_trouver,
lettres_trouvees)

        # A-t-on trouvé le mot, ou nos chances sont-elles épuisées ?
        if mot_a_trouver==mot_trouve:
            print("Félicitation ! Vous avez trouvé le mot
{0}.".format(mot_a_trouver))
        else:
            print("PENDU !!! Vous avez perdu.")

        # On met à jour le score de l'utilisateur
        scores[utilisateur] += nb_chances

        continuer_partie = input("Souhaitez-vous continuer la partie
(O/N) ?")
        continuer_partie = continuer_partie.lower()

    # La partie est finie, on enregistre les scores
    enregistrer_scores(scores)

    # On affiche les scores de l'utilisateur
    print("Vous finissez la partie avec {0}
    points.".format(scores[utilisateur]))
```

Résumé

Vous pouvez voir dans les fonctions demandant à l'utilisateur d'entrer des informations, un bel exemple de **récursivité**. Je ne vais pas détailler ce concept ici, ce n'est pas le sujet, et vous avez pu arriver au même résultat par une autre méthode plus compréhensible pour vous.

Dans l'ensemble, je ne pense pas que le code soit très délicat à comprendre. Vous pouvez vous rendre compte d'à quel point le code du jeu est facile à lire grâce à nos fonctions. On délègue une partie de l'application à nos fonctions qui s'assurent que les choses sont « bien faites ». Si un bug survient, il est plus facile de modifier une fonction que tout un code sans aucune structure.

Par cet exemple, j'espère que vous prendrez bien l'habitude de documenter un maximum vos fichiers et fonctions. C'est réellement un bon réflexe à avoir  .

N'oubliez pas la spécification de l'encodage en tête de chaque fichier, ni la mise en pause du programme sous Windows.

Le moment de détente est terminé… du moins moi, je m'arrête là  . Il est temps de passer aux choses sérieuses… rendez-vous dans la prochaine partie !

Cette partie s'achève ici. Maintenant que vous avez appris à utiliser les objets que Python propose par défaut, il est temps d'apprendre à créer nos premières classes. Rendez-vous dans la prochaine partie !

Partie 3 : La Programmation Orientée Objet, côté développeur

Maintenant que nous avons vu comment utiliser des objets, je crois qu'il est temps d'apprendre à créer nos premières classes, et donc à avoir nos premiers objets personnalisés. Tout au long de cette partie, vous pourrez découvrir les mécanismes qui se cachent derrière les objets que nous avons utilisé (chaîne de caractères, listes, dictionnaires, fichiers...). Je compte même aller un peu plus loin, étant donné l'importance de l'orienté objet en Python.

C'est un point qui n'est pas toujours bien expliqué, d'une importance cruciale. Je vais donc faire de mon mieux pour vous présenter la programmation orientée objet en Python, en accord avec la philosophie du langage.

Première approche des classes

Dans ce chapitre, sans plus attendre, nous allons créer nos premières classes, nos premiers attributs, nos premières méthodes. Mais nous allons essayer de comprendre les mécanismes de la programmation orientée objet en Python. Restez concentrés, ce langage n'a pas fini de vous étonner !

Les classes, tout un monde

Dans la partie précédente, j'avais brièvement décrit les objets comme des variables pouvant contenir elles-mêmes des fonctions et variables. Nous sommes allés plus loin tout au long de la seconde partie, pour découvrir que nos « fonctions contenues dans nos objets » sont appelées des méthodes. En vérité, je me suis cantonné à une définition « pratique » des objets. Alors que derrière la POO (Programmation Orientée Objet) se cache une véritable philosophie .

Pourquoi utiliser des objets ?

Les premiers langages de programmation n'incluaient pas l'orienté objet. Le langage C, pour ne citer que lui, n'utilise pas ce concept et il aura fallu attendre le C++ pour utiliser la puissance de l'orienté objet dans une syntaxe proche de celle du C.

Java, un langage apparu à peu près en même temps que Python, définit une philosophie assez différente de celle du C++ : contrairement à ce dernier, pour coder en Java, il est nécessaire de tout ranger dans des classes. Même l'application standard *Hello World* est contenue dans une classe.

En Python, la liberté est plus grande. Après tout, vous avez pu passer une partie de ce tutoriel sans connaître la façade objet de Python. Et pourtant, le langage Python est totalement orienté objet : tout est objet en Python, vous n'avez pas oublié ? Quand vous croyez utiliser une bête variable, un module, une fonction…, ce sont des objets qui se cachent derrière .

Loin de moi l'idée de faire un comparatif entre différents langages. Ce que je tiens à signaler à votre attention, c'est que plusieurs langages intègrent l'orienté objet, chacun avec une philosophie distincte. Autrement dit, si vous avez appris l'orienté objet dans un autre langage, tel que le C++ ou le Java, ne prenez pas pour acquis que vous allez retrouver les même mécanismes et surtout, la même philosophie. Gardez autant que possible l'esprit dégagé de tout préjugé sur la philosophie objet de Python .

Pour l'instant, nous n'avons donc vu qu'un aspect technique de l'objet. J'irais jusqu'à dire que ce qu'on a vu jusqu'ici, ce n'était qu'une façon « un peu plus esthétique » de coder : il est plus simple et plus compréhensible d'écrire *ma_liste.append(5)* que *append_to_list(ma_liste, 5)*. Mais derrière la POO, il n'y a pas qu'un souci esthétique, loin de là.

Choix du modèle

Bon, comme vous vous en souvenez sûrement (du moins, j'espère), une classe est un peu un modèle sur lequel on va créer des objets. C'est dans la classe qu'on va définir nos méthodes et attributs, les attributs étant des variables contenues dans notre objet.

Mais qu'allons-nous modéliser ? L'orienté objet est plus qu'utile dès lors que l'on s'en sert pour modéliser, représenter, des données un peu plus complexes qu'un simple nombre, ou qu'une chaîne de caractères. Bien sûr, il existe des classes que Python définit pour nous : les nombres, les chaînes et les listes en font partie. Mais on serait bien limité si on ne pouvait faire nos propres classes.

Pour l'instant, nous allons modéliser… une personne . C'est le premier exemple qui me soit venu à l'esprit, nous verrons bien d'autres exemples avant la fin de la partie.

Convention de nommage

Loin de moi l'idée de compliquer l'exercice, mais si on se réfère à la PEP 8 de Python, il est préférable d'utiliser pour des noms de classe la convention dite **Camel Case**.

Cette convention n'utilise pas le signe souligné _ pour séparer les mots. Le principe est de mettre chaque lettre débutant un mot par une majuscule. Par exemple : *MaClasse*.

C'est donc cette convention que je vais utiliser pour les noms de classe. Libre à vous d'en changer, encore une fois rien n'est imposé.

Pour définir une nouvelle classe, on utilise le mot-clé *class*.

Sa syntaxe est assez intuitive : `class NomDeLaClasse:` .

N'exécutez pas encore ce code, nous ne savons pas comment définir nos attributs et nos méthodes.

Petit exercice de modélisation : que va-t-on trouver dans les caractéristiques d'une personne ? Beaucoup de choses, vous en conviendrez. On ne va en retenir que quelques-unes : le nom, le prénom, l'âge, le lieu de résidence… allez, ça suffira.

Ça nous fait donc quatre attributs. Ce sont les variables internes à notre objet qui vont le caractériser. Une personne telle que nous la modélisons sera caractérisée par son nom, son prénom, son âge et son lieu de résidence.

Pour définir les attributs de notre objet, il faut définir un constructeur dans notre classe. Voyons cela de plus près.

Nos premiers attributs

Nous avons défini les attributs qui allaient caractériser notre objet de classe *Personne*. Maintenant, il faut définir dans notre classe une méthode spéciale, appelée un **constructeur**, qui est appelée invariablement quand on souhaite créer un objet depuis notre classe.

Concrètement, un constructeur est une méthode de notre objet se chargeant de créer nos attributs. En vérité c'est même la méthode qui sera appelée quand on voudra créer notre objet.

Voyons le code, ce sera plus parlant :

Code : Python

```
class Personne: # Définition de notre classe Personne
    """Classe définissant une personne caractérisée par :
- son nom
- son prénom
- son âge
- son lieu de résidence

"""
    def __init__(self): # notre méthode constructeur
        """Pour l'instant, on ne va définir qu'un seul attribut"""
        self.nom = "Dupont"
```

Voyons en détail :

- D'abord, la définition de la classe. Elle est constituée du mot-clé *class*, du nom de la classe et des rituels deux points « : »
- Une *docstring* commentant la classe. Encore une fois, c'est une excellente habitude à prendre et je vous encourage à le faire systématiquement. Ce pourra être plus qu'utile quand vous vous lancerez dans de grands projets, d'autant à plusieurs.
- La définition de notre constructeur. Comme vous le voyez, il s'agit d'une définition presque « classique » d'une fonction. Elle a pour nom *__init__*, c'est invariable : tous les constructeurs en Python s'appellent ainsi. Nous verrons plus tard que les noms de méthodes entourés de part et d'autre de deux signes soulignés *(__nommethode__)* sont des méthodes spéciales. Notez que dans notre définition de méthode, on passe un premier paramètre nommé *self*.
- Une nouvelle *docstring*. Je ne complique pas inutilement, je précise donc qu'on va simplement définir un seul attribut pour l'instant dans notre constructeur.
- Dans notre constructeur, nous trouvons l'instanciation de notre attribut *nom*. On crée une variable *self.nom* et on lui donne comme valeur *Dupont*. Je vais détailler un peu plus bas ce qui se passe ici.

Avant tout, pour voir le résultat en action, essayons de créer un objet issu de notre classe :

Code : Python Console

```
>>> bernard = Personne()
>>> bernard
<__main__.Personne object at 0x00B42570>
>>> bernard.nom
'Dupont'
>>>
```

Quand on demande à l'interpréteur d'afficher directement notre objet *bernard*, il nous sort quelque chose d'un peu imbuvable… Bon, l'essentiel est la mention précisant la classe dont l'objet est issu. On peut donc vérifier que c'est bien notre classe *Personne* dont est issu notre objet. On essaye ensuite d'afficher l'attribut *nom* de notre objet *bernard* et on obtient *'Dupont'* (la valeur définie dans notre constructeur). Notez qu'on utilise le point (.), encore et toujours utilisé pour une relation d'appartenance (*nom* est un attribut de l'objet *bernard*). Encore un peu d'explications :

Quand on créée notre objet…

Quand on entre *Personne()*, on appelle le constructeur de notre classe *Personne*, d'une façon quelque peu indirecte que je ne détaillerai pas ici. Celui-ci prend en paramètre une variable un peu mystérieuse : *self*. En fait, il s'agit tout bêtement de notre objet en train de se créer. On écrit dans cet objet l'attribut *nom* le plus simplement du monde : `self.nom = "Dupont"` . À la fin de l'appel au constructeur, Python retourne notre objet *self* modifié, avec notre attribut. On va réceptionner le tout dans notre variable *bernard*.

Si ce n'est pas très clair, pas de panique ! Vous pouvez vous contenter de vous familiariser avec la syntaxe du constructeur Python qui sera souvent la même, et laisser l'aspect un peu théorique de côté, pour plus tard. Nous aurons l'occasion d'y revenir avant la fin du chapitre.

Étoffons un peu notre constructeur

Bon, on avait dit quatre attributs, on n'en a fait qu'un. Et puis notre constructeur pourrait éviter de donner toujours les mêmes valeurs par défaut à chaque fois, tout de même !

C'est juste. Dans un premier temps, on va se contenter de définir nos autres attributs, le prénom, l'âge, le lieu de résidence. Essayez de le faire, normalement vous ne devriez éprouver aucune difficulté.

Voici le code, au cas où :

Code : Python

```
class Personne:
    """Classe définissant une personne caractérisée par :
- son nom
- son prénom
- son âge
- son lieu de résidence

"""
    def __init__(self): # notre méthode constructeur
        """Constructeur de notre classe. Chaque attribut va être
instancié
avec une valeur par défaut… originale.

"""
        self.nom = "Dupont"
        self.prenom = "Jean" # quelle originalité
        self.age = 33 # ça n'engage à rien
```

```
            self.lieu_residence = "Paris"
```

Ça vous paraît évident ? Encore un petit code d'exemple :

Code : Python Console

```
>>> jean = Personne()
>>> jean.nom
'Dupont'
>>> jean.prenom
'Jean'
>>> jean.age
33
>>> jean.lieu_residence
'Paris'
>>> # Jean déménage…
... jean.lieu_residence = "Berlin"
>>> jean.lieu_residence
'Berlin'
>>>
```

Je sens un courant d'air… les habitués de l'objet, une minute .

Cet exemple me paraît assez clair, sur le principe de définition des attributs, accès aux attributs d'un objet créé, modification des attributs d'un objet.

Une toute petite explication en ce qui concerne la ligne 11 : dans beaucoup de tutoriels, on déconseille de modifier un attribut d'objet comme on vient de le faire, en faisant simplement *objet.attribut = valeur*. Si vous venez d'un autre langage, vous pourrez avoir entendu parler des accesseurs et mutateurs. Ces concepts sont repris dans certains tutoriels Python. Mais ils n'ont pas précisément lieu d'être dans ce langage. Tout ça, je le détaillerai dans le chapitre suivant. Pour l'instant, qu'il vous suffise de savoir que quand vous voulez modifier un attribut d'un objet, vous faites *objet.attribut = nouvelle_valeur*. Nous verrons les cas particuliers plus loin.

Bon. Il nous reste encore à faire un constructeur un peu plus intelligent. Pour l'instant, quel que soit l'objet créé, il possède le même nom, prénom, âge et lieu de résidence. On peut les modifier par la suite, bien entendu, mais on peut aussi faire en sorte que le constructeur prenne plusieurs paramètres, disons... le nom et le prénom, pour commencer.

Code : Python

```
class Personne:
    """Classe définissant une personne caractérisée par :
- son nom
- son prénom
- son âge
- son lieu de résidence

"""
    def __init__(self, nom, prenom):
        """Constructeur de notre classe."""
        self.nom = nom
        self.prenom = prenom
        self.age = 33
        self.lieu_residence = "Paris"
```

Et en image :

Code : Python Console

```
>>> bernard = Personne("Micado", "Bernard")
>>> bernard.nom
'Micado'
>>> bernard.prenom
'Bernard'
>>> bernard.age
33
>>>
```

N'oubliez pas que le premier paramètre doit être *self*. En dehors de ça, un constructeur est une fonction plutôt classique, vous pouvez définir des paramètres, par défaut ou non, nommés ou non. Quand vous voudrez créer votre objet, vous appellerez le nom de la classe en passant entre parenthèses les paramètres à entrer. Faites quelques tests, avec plus ou moins de paramètres, je pense que vous saisirez très rapidement le principe.

Attributs de classe

Dans les exemples que nous avons vus jusqu'à présent, nos attributs sont contenus dans notre objet. Ils sont propres à l'objet : si vous créez plusieurs objets, les attributs *nom*, *prenom*, … de chacun ne seront pas forcément identiques d'un objet à l'autre. Mais on peut aussi définir des attributs dans notre classe. Voyons un exemple :

Code : Python

```
class Compteur:
    """Cette classe possède un attribut de classe qui s'incrémente
à chaque
fois que l'on crée un objet de ce type.

"""
    objets_crees = 0 # le compteur vaut 0 au départ
    def __init__(self):
        """À chaque fois qu'on créée un objet, on incrémente le
compteur."""
        Compteur.objets_crees += 1
```

On définit notre attribut de classe directement dans le corps de la classe, sous la définition et la *docstring*, avant la définition du constructeur. Quand on veut l'appeler dans le constructeur, on préfixe le nom de l'attribut de classe par le nom de la classe. Et on y accède de cette façon également, en dehors de la classe. Voyez plutôt :

Code : Python Console

```
>>> Compteur.objets_crees
0
>>> a = Compteur() # on crée un premier objet
>>> Compteur.objets_crees
1
>>> b = Compteur()
>>> Compteur.objets_crees
2
>>>
```

À chaque fois qu'on crée un objet de type *Compteur*, l'attribut de classe *objets_crees* s'incrémente de 1. Ce peut être utile d'avoir des attributs de classe, quand tous nos objets doivent avoir certaines données identiques. Nous aurons l'occasion d'en reparler par la suite  .

Les méthodes, la recette

Les attributs sont des variables propres à notre objet, qui servent à le caractériser. Les méthodes sont plutôt des actions, comme nous l'avons vu dans la partie précédente, agissant sur l'objet. Par exemple, la méthode *append* de la classe *list* permet d'ajouter un élément dans l'objet *list* manipulé.

Pour créer nos premières méthodes, nous allons modéliser… un tableau  . Un tableau noir, oui c'est très bien.

Notre tableau va posséder une surface (un attribut) sur lequel on pourra écrire, que l'on pourra lire et effacer. Pour créer notre classe *TableauNoir* et notre attribut *surface*, vous ne devriez pas avoir de problème :

Code : Python

```
class TableauNoir:
    """Classe définissant une surface sur laquelle on peut écrire,
que l'on peut lire et effacer, par jeu de méthodes. L'attribut
modifié
est 'surface'.

"""
    def __init__(self):
        """Par défaut, notre surface est vide."""
        self.surface = ""
```

Nous avons déjà créé une méthode, aussi vous ne devriez pas être trop surpris par la syntaxe que nous allons voir. Notre constructeur est une méthode en effet, elle en garde la syntaxe. Nous allons donc écrire notre méthode *ecrire* pour commencer.

Code : Python

```
class TableauNoir:
    """Classe définissant une surface sur laquelle on peut écrire,
que l'on peut lire et effacer, par jeu de méthodes. L'attribut
modifié
est 'surface'.

"""
    def __init__(self):
        """Par défaut, notre surface est vide."""
        self.surface = ""
    def ecrire(self, message_a_ecrire):
        """Méthode permettant d'écrire sur la surface du tableau.
Si la surface n'est pas vide, on saute une ligne avant de rajouter
le message à écrire.

"""
        if self.surface != "":
            self.surface += "\n"
        self.surface += message_a_ecrire
```

Passons aux tests :

Code : Python Console

```
>>> tab = TableauNoir()
>>> tab.surface
''
>>> tab.ecrire("Coooool ! Ce sont les vacances !")
>>> tab.surface
"Coooool ! Ce sont les vacances !"
>>> tab.ecrire("Joyeux Noël !")
>>> tab.surface
"Coooool ! Ce sont les vacances !\nJoyeux Noël !"
>>> print(tab.surface)
Coooool ! Ce sont les vacances !
Joyeux Noël !
>>>
```

Notre méthode *ecrire* se charge d'écrire sur notre surface, en rajoutant un saut de ligne pour séparer chaque message.

On retrouve notre paramètre *self* ici. Il est temps de voir un peu plus en détail à quoi il sert.

Le paramètre self

Dans nos méthodes d'objet, qu'on appelle également des **méthodes d'instance**, on trouve dans la définition ce paramètre *self*. L'heure est venue de comprendre ce qu'il signifie :

Une chose qui a son importance : quand vous créez un nouvel objet, ici un tableau noir, les attributs de l'objet sont propres à l'objet créé. C'est logique : si vous créez plusieurs tableaux noirs, ils ne vont pas tous avoir la même surface. Donc les attributs sont contenus dans l'objet.

En revanche, les méthodes, elles, sont contenues dans la classe qui définit notre objet. C'est très important. Quand vous entrez *tab.ecrire(...)*, Python va chercher la méthode *ecrire* non pas dans l'objet *tab*, mais dans la classe *TableauNoir*.

Code : Python Console

```
>>> tab.ecrire
<bound method TableauNoir.ecrire of <__main__.TableauNoir object at
0x00B3F3F0>>
>>> TableauNoir.ecrire
<function ecrire at 0x00BA5810>
>>> help(TableauNoir.ecrire)
Help on function ecrire in module __main__:
ecrire(self, message_a_ecrire)
Méthode permettant d'écrire sur la surface du tableau.
Si la surface n'est pas vide, on saute une ligne avant de rajouter
le message à écrire.
>>> TableauNoir.ecrire(tab, "essai")
>>> tab.surface
'essai'
>>>
```

Comme vous le voyez, quand vous entrez *tab.ecrire(...* ça revient au même que si vous écrivez *TableauNoir.ecrire(tab, ...)*. Votre paramètre *self*, c'est l'objet qui appelle la méthode. C'est pour cette raison que vous modifiez la surface de l'objet en appelant *self.surface*.

Pour résumer, quand vous devez travailler dans une méthode de l'objet sur l'objet lui-même, vous allez passer par *self*.

Le nom *self* est une très forte convention de nommage. Je vous déconseille de changer ce nom. Certains programmeurs, trouvant qu'écrire *self* à chaque fois est excessivement long, l'abrègent en une unique lettre *s*. Évitez ce raccourci. D'une manière générale, évitez de changer le nom. Une méthode d'objet travaille avec le paramètre *self*.

N'est-ce pas effectivement plutôt long, de devoir toujours travailler avec *self* à chaque fois qu'on souhaite faire appel à l'objet ?

Ça peut le sembler, oui. C'est d'ailleurs l'un des reproches qu'on fait au langage Python. Certains langages travaillent implicitement sur les attributs et méthodes d'un objet sans avoir besoin de les appeler spécifiquement. Mais c'est moins clair, et peut susciter la confusion. En Python, dès qu'on voit *self*, on sait que c'est un attribut ou une méthode interne à l'objet qui va être appelé.

Bon, voyons nos autres méthodes. Nous devons encore coder *lire* qui va se charger d'afficher notre surface, et *effacer* qui va effacer le contenu de notre surface. Si vous avez compris ce qui vient de suivre, vous devriez écrire ces méthodes sans aucun problème, elles sont très simples. Sinon, n'hésitez pas à relire, jusqu'à ce que le déclic se fasse.

Code : Python

```
class TableauNoir:
```

```
    """Classe définissant une surface sur laquelle on peut écrire,
que l'on peut lire et effacer, par jeu de méthodes. L'attribut
modifié
est 'surface'.

"""
    def __init__(self):
        """Par défaut, notre surface est vide."""
        self.surface = ""
    def ecrire(self, message_a_ecrire):
        """Méthode permettant d'écrire sur la surface du tableau.
Si la surface n'est pas vide, on saute une ligne avant de rajouter
le message à écrire.

"""
        if self.surface != "":
            self.surface += "\n"
        self.surface += message_a_ecrire
    def lire(self):
        """Cette méthode se charge tout bêtement d'afficher, grâce
à print,
la surface du tableau.

"""
        print(self.surface)
    def effacer(self):
        """Cette méthode permet d'effacer la surface du tableau."""
        self.surface = ""
```

Et encore une fois, le code de test :

Code : Python Console

```
>>> tab = TableauNoir()
>>> tab.lire()
>>> tab.ecrire("Salut tout le monde.")
>>> tab.ecrire("La forme ?")
>>> tab.lire()
Salut tout le monde.
La forme ?
>>> tab.effacer()
>>> tab.lire()
>>>
```

Voilà ! Tout est bon. Et avec nos méthodes bien documentées, un petit coup de `help(TableauNoir)` et vous obtenez une belle description de l'utilité de votre classe. C'est très pratique, n'oubliez pas les docstrings  .

Des méthodes de classe

Comme on trouve des attributs propres à la classe, on trouve aussi des méthodes de classe, qui ne sont donc pas appelable depuis l'objet mais depuis la classe. C'est un peu plus rare, mais ça peut être utile parfois. Notre méthode de classe se définit exactement comme une méthode d'objet, à la différence qu'elle ne prend pas en premier paramètre *self*, puisqu'elle ne travaille sur aucun objet, mais sur la classe. Reprenons notre exemple de tout à l'heure :

Code : Python

```
class Compteur:
    """Cette classe possède un attribut de classe qui s'incrémente
à chaque
fois que l'on créée un objet de ce type.

"""
```

```
    objets_crees = 0 # le compteur vaut 0 au départ
    def __init__(self):
        """A chaque fois qu'on créée un objet, on incrémente le
compteur."""
        Compteur.objets_crees += 1
    def combien():
        """Méthode de classe affichant combien d'objets ont été
créés."""
        print("Jusqu'à présent, {0} objets ont été créés.".format( \
                Compteur.objets_crees))
```

Là encore, pour l'appeler, on ne passe pas par l'objet mais par le nom de la classe :

Code : Python Console

```
>>> Compteur.combien()
Jusqu'à présent, 0 objets ont été créés.
>>> a = Compteur()
>>> Compteur.combien()
Jusqu'à présent, 1 objets ont été créés.
>>> b = Compteur()
>>> Compteur.combien()
Jusqu'à présent, 2 objets ont été créés.
>>>
```

Si vous vous emmêlez un peu avec les attributs et méthodes de classe, ce n'est pas bien grave. Retenez surtout les attributs et méthodes d'objet, c'est surtout sur ceux-ci que je me suis attardé et c'est ceux que vous retrouverez la plupart du temps .

Rappel

Les noms de méthode encadrés par deux soulignés de part et d'autre sont des **méthodes spéciales**. Ne nommez pas vos méthodes ainsi. Nous découvrirons plus tard ces méthodes particulières.

Exemple de nom de méthode à éviter : *__mamethode__*.

Un peu d'introspection

Encore de la philosophie ?

Et bien... le terme d'**introspection**, je le reconnais, fait penser à quelque chose de plutôt abstrait . Et pourtant, vous allez très vite comprendre l'idée derrière : Python propose plusieurs techniques pour explorer un objet, connaître ses méthodes ou attributs.

Quel est l'intérêt ? Quand on développe une classe, on sait généralement ce qu'il y a dedans non ?

En effet. L'utilité, à notre niveau, ne saute pas encore aux yeux. Et c'est pour cela que je ne vais pas trop m'attarder dessus. Si vous ne voyez pas l'intérêt, contentez-vous de garder dans un coin de votre tête les deux techniques que nous allons voir. Arrivera un jour où vous pourrez y trouver une utilité . Pour l'heure donc, voyons plutôt l'effet :

La fonction dir

La première technique d'introspection que nous allons voir est la fonction *dir*. Elle prend en paramètre un objet et retourne la liste de ses attributs et méthodes.

Code : Python

```
class Test:
    """Une classe de test tout simplement"""
    def __init__(self):
        """On définit dans notre constructeur un unique attribut"""
        self.mon_attribut = "ok"

    def afficher_attribut(self):
        """Méthode affichant l'attribut 'mon_attribut'"""
        print("Mon attribut est {0}.".format(self.mon_attribut))
```

Code : Python Console

```
>>> # Créons un objet de la classe Test
... un_test = Test()
>>> un_test.afficher_attribut()
Mon attribut est ok.
>>> dir(un_test)
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__',
'__format__', '__g
e__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__le__', '__lt__',
'__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '_
_setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'affich
er_attribut', 'mon_attribut']
>>>
```

La fonction *dir* retourne une liste comprenant le nom des attributs et méthodes de l'objet qu'on lui passe en paramètre. Vous pouvez remarquer que tout est mélangé, c'est normal : pour Python, les méthodes, les fonctions, les classes, les modules sont des objets. Ce qui différencie en premier lieu une variable d'une fonction, c'est qu'une fonction est exécutable (**callable**). La fonction *dir* se contente de retourner tout ce qu'il y a dans l'objet, sans distinction.

Euh, c'est quoi tout ça ? On a jamais défini toutes ces méthodes ou attributs !

Non, en effet. Nous verrons plus loin qu'il s'agit de **méthodes spéciales** utiles à Python.

L'attribut spécial __dict__

Par défaut, quand vous développez une classe, tous les objets construits depuis cette classe posséderont un attribut spécial *__dict__*. Cet attribut est un dictionnaire qui contient le nom des attributs en clé et leur valeur correspondante en valeur.

Voyez plutôt :

Code : Python Console

```
>>> un_test = Test()
>>> un_test.__dict__
{'mon_attribut': 'ok'}
>>>
```

Pourquoi "attribut spécial" ?

C'est un attribut un peu particulier car ce n'est pas vous qui le créez, c'est Python. Il est entouré de deux signes soulignés __ de part et d'autre, ce qui signifie qu'il a une signification pour Python et n'est pas un attribut "standard". Vous verrez plus loin dans ce cours des **méthodes spéciales** qui reprennent la même syntaxe.

Peut-on modifier ce dictionnaire ?

Vous pouvez. Et en modifiant la valeur de l'attribut, vous modifiez l'attribut dans l'objet également.

Code : Python Console

```
>>> un_test.__dict__["mon_attribut"] = "plus ok"
>>> un_test.afficher_attribut()
Mon attribut est plus ok.
>>>
```

D'une manière générale, ne faites appel à l'introspection que si vous avez une bonne raison de le faire et évitez ce genre de syntaxe. Il est quand même plus propre de faire *objet.attribut = valeur* que *objet.__dict__[nom_attribut] = valeur*  .

Nous n'irons pas plus loin dans ce chapitre. Je pense que vous découvrirez dans la suite de ce tutoriel l'utilité des deux méthodes que je vous ai montré  .
J'espère que cette première approche des classes vous a plu. Dans le prochain chapitre, nous allons regarder le mécanisme des propriétés et parler de l'encapsulation, pour éviter les pièges que l'on peut faire en Python. N'hésitez pas à tenter de modéliser grâce à des classes d'autres objets, ceux qui vous passent par la tête (un four à micro-onde, un fauteuil, un escalier, une chaussette...) l'océan des objets s'ouvre à vous à présent  . Évitez de passer au chapitre suivant s'il reste des zones d'ombre dans votre esprit. Prenez le temps de relire ce chapitre si cela est nécessaire.

Les propriétés

Dans le chapitre précédent, nous avons appris à créer nos premiers attributs et méthodes. Mais nous avons encore assez peu parlé de la philosophie objet. Il existe quelques confusions que je vais éclaircir de mon mieux. Nous allons découvrir dans ce chapitre les propriétés, un concept propre à Python et à quelques autres langages tel le Ruby. C'est une fonctionnalité qui, à elle seule, change l'approche objet et le principe d'encapsulation. Nous allons découvrir tout cela ici .

Que dit l'encapsulation ?

L'encapsulation est un principe qui consiste à cacher ou protéger certaines données de notre objet. Dans la plupart des langages orientés objet, tels le C++, le Java ou le PHP, on va considérer que nos attributs d'objet ne doivent pas être accessibles depuis l'extérieur de la classe. Autrement dit, vous n'avez pas le droit de faire, depuis l'extérieur de la classe, *mon_objet.mon_attribut*.

Mais c'est stupide ! Comment on fait pour accéder aux attributs ?

On va définir des méthodes un peu particulières, appelées des **accesseurs** et **mutateurs**. Les accesseurs donnent accès à l'attribut. Les mutateurs permettent de le modifier. Concrètement, au lieu d'entrer *mon_objet.mon_attribut*, vous allez entrer *mon_objet.get_mon_attribut()* (get signifie « récupérer », c'est le préfixe généralement utilisé pour un accesseur). Et pour modifier l'attribut, vous n'allez pas entrer *mon_objet.mon_attribut = valeur* mais *mon_objet.set_mon_attribut(valeur)* (set signifie dans ce contexte « modifier », c'est le préfixe usuel pour un mutateur).

C'est bien tordu tout ça ! Pourquoi ne peut-on pas accéder aux attributs directement, comme on l'a fait dans le chapitre précédent ?

Ah mais d'abord, je n'ai pas dit que vous ne pouviez pas. Vous pouvez très bien accéder aux attributs d'un objet directement, comme on l'a fait dans le chapitre précédent. Je ne fais ici que résumer le principe d'encapsulation tel qu'on peut le trouver dans d'autres langages. En Python, c'est un peu plus subtil.

Mais pour répondre à la question, il peut être très pratique de « sécuriser » certaines données de notre objet. Par exemple, faire en sorte qu'un attribut de notre objet ne soit pas modifiable. Ou alors, que dès que l'on modifie un attribut, ça mette à jour un autre attribut. Les cas sont multiples, et c'est très utile de pouvoir contrôler l'accès en lecture ou en écriture sur certains attributs de notre objet.

L'inconvénient de devoir écrire des accesseurs et mutateurs, comme vous l'aurez sans doute compris, c'est qu'il faut créer deux méthodes pour chaque attribut de notre classe. D'abord, c'est assez lourd. Ensuite nos méthodes se ressemblent plutôt. Certains environnements de développement proposent, il est vrai, de créer ces accesseurs et mutateurs pour nous, automatiquement. Mais ça ne résout pas vraiment le problème, vous en conviendrez .

Python a une philosophie un peu différente : pour tous les objets dont on n'attend pas une action particulière, on va y accéder directement, comme nous l'avons fait dans le chapitre précédent. On peut y accéder et les modifier en entrant simplement *mon_objet.mon_attribut*. Et pour certains, on va créer des propriétés.

Les propriétés à la casserole

Pour commencer, une petite précision : en C++ ou en Java par exemple, dans notre définition de classe, on va préciser des principes d'accès qui vont dire si l'attribut (ou le groupe d'attributs) est privé ou public. Pour schématiser, si l'attribut est public, on peut y accéder depuis l'extérieur de la classe et le modifier. S'il est privé, on ne peut pas. On doit passer par des accesseurs ou mutateurs.

En Python, il n'y a pas d'attributs privés. Tout est public. Cela signifie que si vous voulez modifier un attribut depuis l'extérieur de la classe, vous le pouvez. Pour faire respecter l'encapsulation propre au langage, on va la fonder sur des conventions que nous allons découvrir un peu plus bas, mais surtout sur le bon sens de l'utilisateur de notre classe (à savoir, si j'ai écrit que cet attribut est inaccessible depuis l'extérieur de la classe, ne pas y accéder depuis l'extérieur de la classe).

Les propriétés sont un moyen transparent de manipuler des attributs d'objet. Elles permettent de dire à Python : « Quand un utilisateur souhaite modifier cet attribut, fais cela. » De cette façon, on peut rendre certains attributs tout à fait inaccessibles depuis l'extérieur de la classe, ou dire qu'un attribut ne sera visible qu'en lecture et non modifiable. Ou encore, on peut faire en sorte que si on modifie un attribut, Python recalcule la valeur d'un autre attribut de l'objet.

Pour l'utilisateur, c'est absolument transparent : il croit avoir, dans tous les cas, un accès direct à l'attribut. C'est dans la définition de la classe que vous allez préciser que tel ou tel attribut doit être accessible ou modifiable grâce à certaines propriétés.

Mais ces propriétés, c'est quoi ?

Hum… eh bien je pense que pour le comprendre, il vaut mieux les voir en action . Les propriétés sont des objets de Python un peu particuliers. Elles prennent la place d'un attribut et agissent différemment en fonction du contexte dans lequel elles sont appelées. Si on les appelle pour modifier l'attribut par exemple, elles vont rediriger vers une méthode que nous avons créée, qui gère le cas où « on souhaite modifier l'attribut ». Mais trêve de théorie .

Les propriétés en action

Une propriété ne se crée pas dans le constructeur mais dans le corps de la classe. J'ai dit qu'il s'agissait d'une classe, son nom est *property*. Elle attend quatre paramètres, tous optionnels :

- la méthode donnant accès à l'attribut ;
- la méthode modifiant l'attribut ;
- la méthode appelée quand on souhaite supprimer l'attribut ;
- la méthode appelée quand on demande de l'aide sur l'attribut.

En pratique, on utilise surtout les deux premiers paramètres : ceux définissant la méthode d'accès et de modification, autrement dits notre accesseur et mutateur d'objet.

Mais j'imagine que ce n'est pas très clair dans votre esprit. Considérez le code suivant, je le détaillerai plus bas comme d'habitude :

Code : Python

```
class Personne:
    """Classe définissant une personne caractérisée par :
— son nom ;
— son prénom ;
— son âge ;
— son lieu de résidence.

"""
    def __init__(self, nom, prenom):
        """Constructeur de notre classe."""
        self.nom = nom
        self.prenom = prenom
        self.age = 33
        self._lieu_residence = "Paris" # notez le souligné _ devant
le nom
    def _get_lieu_residence(self):
        """Méthode qui sera appelée quand on souhaite accéder en
lecture
à l'attribut 'lieu_residence'.

"""
        print("On accède à l'attribut lieu_residence !")
        return self._lieu_residence
    def _set_lieu_residence(self, nouvelle_residence):
        """Méthode appelée quand on souhaite modifier le lieu de
résidence."""
        print("Attention, il semble que {0} déménage à {1}.".format(
\
                self.prenom, nouvelle_residence))
        self._lieu_residence = nouvelle_residence
    # On va dire à Python que notre attribut lieu_residence pointe
vers une
    # propriété
    lieu_residence = property(_get_lieu_residence,
_set_lieu_residence)
```

La syntaxe générale de la classe, vous devriez (j'espère) la reconnaître. C'est concernant le lieu de résidence que ça change

un peu :

- Tout d'abord dans le constructeur, on ne crée pas un attribut *self.lieu_residence* mais *self._lieu_residence*. Il n'y a qu'un petit caractère de différence, le signe souligné _ placé en tête du nom de l'attribut. Et pourtant ce signe change beaucoup de choses. La convention veut qu'on n'accède pas, depuis l'extérieur de la classe, à un attribut commençant par un souligné _. C'est une convention, rien ne vous l'interdit… sauf encore une fois le bon sens. 
- On définit une première méthode, commençant elle aussi par un souligné _, nommée *_get_lieu_residence*. C'est la même règle que pour les attributs : on n'accède pas à une méthode commençant par un souligné _ depuis l'extérieur de la classe. Si vous avez compris ma petite explication sur les accesseurs et mutateurs, vous devriez comprendre rapidement à quoi sert cette méthode : elle se contente de retourner le lieu de résidence. Là encore, l'attribut manipulé n'est pas *lieu_residence* mais *_lieu_residence*. Comme on est dans la classe, on a le droit de le manipuler.
- La seconde méthode a la forme d'un mutateur. Elle se nomme *_set_lieu_residence*, doit donc être aussi inaccessible depuis l'extérieur de la classe. À la différence de l'accesseur, elle prend un paramètre : le nouveau lieu de résidence. En effet, c'est une méthode qui doit être appelée quand on cherche à modifier le lieu de résidence, il lui faut donc le nouveau lieu de résidence qu'on souhaite voir attribué à l'objet.
- Enfin, la dernière ligne de la classe est très intéressante. Il s'agit de la définition d'une propriété. On lui dit que l'attribut *lieu_residence* (cette fois, sans signe souligné _) doit être une propriété. On définit dans notre propriété la méthode d'accès (l'accesseur) et de modification (le mutateur), dans l'ordre.

Quand on va vouloir accéder à *objet.lieu_residence*, Python va tomber sur une propriété redirigeant vers la méthode *_get_lieu_residence*. Quand on souhaite modifier la valeur de l'attribut, en entrant *objet.lieu_residence = valeur*, Python va appeler la méthode *_set_lieu_residence* en lui passant en paramètre la nouvelle valeur.

Ce n'est pas clair ? Voyez cet exemple :

Code : Python Console

```
>>> jean = Personne("Micado", "Jean")
>>> jean.nom
'Micado'
>>> jean.prenom
'Jean'
>>> jean.age
33
>>> jean.lieu_residence
On accède à l'attribut lieu_residence !
'Paris'
>>> jean.lieu_residence = "Berlin"
Attention, il semble que Jean déménage à Berlin.
>>> jean.lieu_residence
On accède à l'attribut lieu_residence !
'Berlin'
>>>
```

Notre accesseur et notre mutateur se contentent d'afficher un message, pour bien qu'on se rende compte que ce sont eux qui sont appelés quand on souhaite manipuler l'attribut *lieu_residence*. Vous pouvez aussi ne définir qu'un accesseur, dans ce cas l'attribut ne pourra pas être modifié.

Il est aussi possible de définir en troisième position du constructeur *property* une méthode qui sera appelée quand on fera *del objet.lieu_residence*, et en quatrième position une méthode qui sera appelée quand on fera *help(objet.lieu_residence)*. Ces deux dernières fonctionnalités sont un peu moins utilisées, mais elles existent.

Voilà, vous connaissez à présent la syntaxe pour créer des propriétés. Entraînez-vous, ce n'est pas toujours évident au début. C'est un concept très puissant, il serait dommage de passer à côté.

Résumons le principe d'encapsulation en Python

Je vais condenser un peu tout le chapitre ici. Nous avons vu qu'en Python, quand on souhaitait accéder à un attribut d'un objet, on fait tout bêtement *objet.attribut*. Par contre, on doit éviter d'accéder ainsi à des attributs ou des méthodes commençant par un

signe souligné _, question de convention. Si par hasard une action particulière doit être faite quand on accède à un attribut, pour le lire simplement, pour le modifier, le supprimer…, on fait appel à des propriétés. Pour l'utilisateur de la classe, ça revient au même : il entre toujours *objet.attribut*. Mais dans la définition de notre classe, on fait en sorte que l'attribut visé soit une propriété avec certaines méthodes, accesseurs, mutateurs ou autre, qui définissent ce que Python doit faire quand on souhaite lire, modifier, supprimer l'attribut.

Avec ce concept, on perd beaucoup moins de temps. On ne fait pas systématiquement un accesseur et un mutateur pour chaque attribut, et le code est bien plus lisible. Autant de gagné .

Certaines classes ont besoin qu'un traitement récurrent soit effectué sur leurs attributs. Par exemple, quand je souhaite modifier un attribut de l'objet (n'importe quel attribut), l'objet doit être enregistré dans un fichier. Dans ce cas, on n'utilisera pas les propriétés, qui sont plus utiles pour des cas particuliers, mais plutôt des méthodes spéciales, que nous découvrirons dans le chapitre suivant.
C'est fini ! Vous avez découvert l'encapsulation façon Python qui est un peu différente des autres langages. Malheureusement, beaucoup de tutoriels rédigés pour le Python prennent pour base l'encapsulation des autres langages. Ce n'est absolument pas interdit hein , mais c'est un peu dommage, quand un système tel que les propriétés existe. Vous n'avez peut-être pas encore appréhendé toute la puissance de ce concept, mais nous aurons l'occasion d'y revenir.

Pour l'instant, jetons un œil aux méthodes spéciales, encore un mécanisme puissant (oui oui il en reste beaucoup).

Les méthodes spéciales

Les méthodes spéciales sont des méthodes d'instance que Python reconnaît et sait utiliser, dans certains contextes. Elles peuvent servir à « dire quoi faire à Python » quand il se retrouve devant une expression comme *mon_objet1 + mon_objet2*, voire *mon_objet[indice]*. Et, encore plus fort, elles contrôlent comment un objet se crée et comment accéder à ses attributs.

Bref, encore une fonctionnalité puissante et utile du langage que je vous invite à découvrir. Prenez note du fait que je ne peux pas expliquer dans ce chapitre la totalité des méthodes spéciales. Il y en a qui ne sont pas de notre niveau, il y en a sur lesquelles je passerai plus vite que d'autres. En cas de doute, ou si vous êtes curieux, je vous encourage d'autant plus à aller faire un tour du côté de la documentation de Python, disponible sur le site officiel.

Édition de l'objet et accès aux attributs

Vous avez déjà vu, dès la première partie de ce chapitre, un exemple de **méthode spéciale**. Pour ceux qui ont la mémoire courte, il s'agit de notre constructeur. Une méthode spéciale en Python est entourée de part et d'autre de deux signes souligné (_). Le nom d'une méthode spéciale est donc sous cette forme : *__methodespeciale__*.

Pour commencer, nous allons voir les méthodes qui travaillent directement sur l'objet. Nous verrons ensuite, plus spécifiquement, les méthodes qui permettent d'accéder aux attributs.

Édition de l'objet

Les méthodes que nous allons voir permettent de travailler sur l'objet. Elles interviennent au moment de le créer et au moment de le supprimer. La première, vous devriez la reconnaître : c'est notre constructeur. Elle s'appelle *__init__*, prend un nombre variable d'arguments et permet de contrôler la création de nos attributs.

Code : Python

```
class Exemple:
    """Un petit exemple de classe"""
    def __init__(self, nom):
        """Exemple de constructeur"""
        self.nom = nom
        self.autre_attribut = "une valeur"
```

Pour créer notre objet, on utilise le nom de la classe et on passe, entre parenthèses, les informations qu'attend notre constructeur :

Code : Python

```
mon_objet = Exemple("un premier exemple")
```

J'ai un peu simplifié ce qui se passe, mais pour l'instant c'est ce qu'il y a à retenir. Comme vous l'avez vu ensuite, on peut, dès le moment où on a créé l'objet, accéder à ses attributs grâce à *mon_objet.nom_attribut* et exécuter ses méthodes grâce à *mon_objet.nom_methode(...)*.

Il existe également une autre méthode, *__del__*, qui va être appelée au moment de la destruction de l'objet.

La destruction ? Quand un objet se détruit-il ?

Bonne question. Il y a plusieurs cas : d'abord, quand vous voulez le supprimer explicitement, grâce au mot-clé *del* (`del mon_objet`). Ensuite, si l'espace de nom contenant l'objet est détruit, l'objet l'est également. Par exemple, si vous instanciez l'objet dans un corps de fonction : à la fin de l'appel à la fonction, la méthode *__del__* de l'objet sera appelée. Enfin, si votre objet résiste envers et contre tout pendant l'exécution du programme, il sera supprimé à la fin de l'exécution.

Code : Python

```
    def __del__(self):
        """Méthode appelée quand l'objet est supprimé"""
        print("C'est la fin ! On me supprime !")
```

À quoi ça peut bien servir, de contrôler la destruction d'un objet ?

Souvent, à rien. Python s'en sort comme un grand garçon  , il n'a pas besoin d'aide. Parfois, on peut vouloir récupérer des informations d'état sur l'objet au moment de sa suppression. Mais ce n'est qu'un exemple : les méthodes spéciales sont un moyen d'exécuter des actions personnalisées sur certains objets, dans un cas précis. Si l'utilité ne saute pas aux yeux, vous pourrez en trouver une un beau jour, en codant votre projet  .

Souvenez-vous que si vous ne définissez pas de méthode spéciale pour telle ou telle action, Python aura un comportement par défaut dans le contexte où cette méthode est appelée. Écrire une méthode spéciale permet de modifier ce comportement par défaut. Dans l'absolu, vous n'êtes même pas obligé d'écrire un constructeur, même si c'est un peu plus tordu  .

Représentation de l'objet

Nous allons voir deux méthodes spéciales qui permettent de contrôler comment l'objet est représenté et affiché. Vous avez sûrement déjà pu constater que quand on instancie des objets de nos propres classes, si on essaye de les afficher, directement dans l'interpréteur ou grâce à *print*, on obtient quelque chose assez laid :

Code : Python Console

```
<__main__.XXX object at 0x00B46A70>
```

On a certes les informations utiles, mais pas forcément celles qu'on veut, et l'ensemble n'est pas magnifique, il faut bien le reconnaître  .

La première méthode permettant de remédier à cet état de fait est *__repr__*. Elle affecte la façon dont est affiché l'objet quand on l'entre directement. On la redéfinit quand on souhaite faciliter le *debug* sur certains objets :

Code : Python

```
class Personne:
    """Classe représentant une personne"""
    def __init__(self, nom, prenom):
        """Constructeur de notre classe"""
        self.nom = nom
        self.prenom = prenom
        self.age = 33
    def __repr__(self):
        """Quand on entre notre objet dans l'interpréteur"""
        return "Personne: nom({0}), prénom({1}), âge({2})".format( \
                self.nom, self.prenom, self.age)
```

Et le résultat en image  :

Code : Python Console

```
>>> p1 = Personne("Micado", "Jean")
>>> p1
Personne: nom(Micado), prénom(Jean), âge(33)
>>>
```

Comme vous le voyez, la méthode *__repr__* ne prend aucun paramètre (sauf, bien entendu *self*), et retourne une chaîne de caractères : la chaîne à afficher quand on entre l'objet directement dans l'interpréteur.

On peut obtenir cette chaîne grâce, également, à la fonction *repr* qui se contente d'appeler la méthode spéciale *__repr__* de l'objet passé en paramètre :

Code : Python Console

```
>>> p1 = Personne("Micado", "Jean")
>>> repr(p1)
'Personne: nom(Micado), prénom(Jean), âge(33)'
>>>
```

Il existe une seconde méthode spéciale, *__str__*, spécialement utilisée pour afficher l'objet, avec *print*. Par défaut, si aucune méthode *__str__* n'est définie, Python appelle la méthode *__repr__* de l'objet. La méthode *__str__* est également appelée si vous désirez convertir votre objet en chaîne avec le constructeur *str*.

Code : Python

```
class Personne:
    """Classe représentant une personne"""
    def __init__(self, nom, prenom):
        """Constructeur de notre classe"""
        self.nom = nom
        self.prenom = prenom
        self.age = 33
    def __str__(self):
        """Méthode permettant d'afficher plus joliment notre
objet"""
        return "{0} {1}, âgé de {2} ans".format( \
                self.prenom, self.nom, self.age)
```

Et en pratique :

Code : Python Console

```
>>> p1 = Personne("Micado", "Jean")
>>> print(p1)
Jean Micado, âgé de 33 ans
>>> chaine = str(p1)
>>> chaine
'Jean Micado, âgé de 33 ans'
>>>
```

Accès aux attributs de notre objet

Nous allons découvrir trois méthodes permettant de définir comment accéder et modifier nos attributs.

La méthode __getattr__

La méthode spéciale *__getattr__* permet de définir une méthode d'accès à nos attributs plus large que celle que Python propose par défaut. En fait, cette méthode est appelée quand vous entrez *objet.attribut* (non pas pour modifier l'attribut, simplement pour y accéder). Python recherche l'attribut et, s'il ne le trouve pas dans l'objet, et si une méthode *__getattr__* existe, il va l'appeler en passant en paramètre le nom de l'attribut recherché, sous la forme d'une chaîne de caractères.

Un petit exemple ?

Code : Python Console

```
>>> class Protege:
...     """Classe possédant une méthode d'accès à ses attributs
particulière :
... Si l'attribut n'est pas trouvé, on affiche une alerte et
retourne None.
...
... """
...     def __init__(self):
...         """On crée quelques attributs par défaut"""
...         self.a = 1
...         self.b = 2
...         self.c = 3
...     def __getattr__(self, nom):
...         """Si Python ne trouve pas l'attribut nommé nom, il
appelle
... cette méthode. On affiche une alerte.
...
... """
...         print("Alerte ! Il n'y a pas d'attribut {0} ici
!".format(nom))
...
>>> pro = Protege()
>>> pro.a
1
>>> pro.c
3
>>> pro.e
Alerte ! Il n'y a pas d'attribut e ici !
>>>
```

Vous comprenez le principe ? Si l'attribut auquel on souhaite accéder existe, notre méthode n'est pas appelée. En revanche, si l'attribut n'existe pas, notre méthode *__getattr__* est appelée. On lui passe en paramètre le nom de l'attribut auquel Python essaye d'accéder. Ici, on se contente d'afficher une alerte. Mais on pourrait tout aussi bien rediriger vers un autre attribut. Par exemple, si on essaye d'accéder à un attribut qui n'existe pas, on redirige vers *self.c*. Je vous laisse faire l'essai, ça n'a rien de difficile  .

La méthode __setattr__

Cette méthode définit l'accès à un attribut destiné à être modifié. Si vous entrez *objet.nom_attribut = nouvelle_valeur*, la méthode spéciale *__setattr__* sera appelée ainsi : `objet.__setattr__("nom_attribut", nouvelle_valeur)`. Là encore, le nom de l'attribut recherché est passé sous la forme d'une chaîne de caractères. Cette méthode permet de faire une action dès qu'un attribut est modifié, par exemple enregistrer l'objet :

Code : Python

```
def __setattr__(self, nom_attr, val_attr):
        """Méthode appelée quand on fait objet.nom_attr = val_attr.
On se charge d'enregistrer l'objet.

"""
        object.__setattr__(self, nom_attr, val_attr)
        self.enregistrer()
```

Une explication s'impose concernant la ligne *6*, je pense. Je vais faire de mon mieux, sachant que j'expliquerai bien plus en détail, dans le chapitre suivant, le concept d'héritage. Pour l'instant, il vous suffit de savoir que toutes les classes que nous créons sont héritées de la classe *object*. Cela veut dire essentiellement qu'elles reprennent les mêmes méthodes. La classe *object* est définie par Python. Quand je disais plus haut que, si vous ne définissiez pas une certaine méthode spéciale, Python avait un

comportement par défaut, ce comportement est défini par la classe *object*.

Toutes les méthodes spéciales sont déclarées dans *object*. Si vous faites par exemple *objet.attribut = valeur* sans avoir défini, dans votre classe, de méthode *__setattr__*, c'est la méthode *__setattr* de la classe *object* qui sera appelée.

Mais si vous redéfinissez la méthode *__setattr__* dans votre classe, ce sera celle que vous définissez, et non celle de *object* qui sera appelée. Oui mais… vous ne savez pas comment Python fait, réellement, pour modifier la valeur d'un attribut. Le mécanisme derrière la méthode vous est inconnu.

Si vous essayez, dans la méthode *__setattr__*, de faire quelque chose comme *self.attribut = valeur*, vous allez créer une jolie erreur : Python va vouloir modifier un attribut, il appelle *__setattr__* de la classe que vous avez définie, il tombe dans cette méthode sur nouvelle affectation d'attribut, il rappelle donc de nouveau *__setattr__*... et tout ça, jusqu'à l'infini ou presque. Python met une protection pour éviter qu'une méthode ne s'appelle elle-même à l'infini, mais ça ne règle pas le problème  .

Tout ça pour dire que dans votre méthode *__setattr__*, vous ne pouvez modifier d'attribut de la façon que vous connaissez. Si vous le faites, *__setattr__* appellera *__setattr__* qui appellera *__setattr__*… à l'infini. Donc si on souhaite modifier un attribut, on va se référer à la méthode *__setattr__* définie dans la classe *object*, la classe mère dont toutes nos classes héritent.

Si toutes ces explications vous ont paru plutôt dures, ne vous en faites pas trop : je détaillerai dans le chapitre suivant ce qu'est l'héritage, vous comprendrez sûrement mieux à ce moment  .

La méthode __delattr__

Cette méthode spéciale est appelée quand on souhaite supprimer un attribut de l'objet, en faisant `del objet.attribut` par exemple. Elle prend en paramètre, outre *self*, le nom de l'attribut que l'on souhaite supprimer. Voici un exemple d'une classe dont on ne peut supprimer aucun attribut :

Code : Python

```
def __delattr__(self, nom_attr):
        """On ne peut supprimer d'attributs, on lève l'exception
AttributeError

"""
        raise AttributeError("Vous ne pouvez supprimer aucun
attribut de cette classe")
```

Là encore, si vous voulez supprimer un attribut, n'utilisez pas dans votre méthode *del self.attribut*. Sinon, vous allez mettre Python en colère  . Passez par *object.__delattr__* qui sait mieux que nous comment tout cela fonctionne.

Un petit bonus

Voici quelques fonctions qui font à peu près ce que nous avons fait, mais en utilisant des chaînes de caractères pour nom d'attribut. Vous pourrez en avoir l'usage :

Code : Python Console

```
objet = MaClasse() # On crée une instance de notre classe
getattr(objet, "nom") # semblable à objet.nom
setattr(objet, "nom", val) # = objet.nom = val ou
objet.__setattr__("nom", val)
delattr(objet, "nom") # = del objet.nom ou objet.__delattr__("nom")
hasattr(objet, "nom") # renvoie True si l'attribut "nom" existe,
False sinon
```

Peut-être ne voyez-vous pas trop l'intérêt de ces fonctions qui prennent toutes, en premier paramètre, l'objet sur lequel travailler et en second le nom de l'attribut (sous la forme d'une chaîne), mais ce peut être très pratique parfois de travailler plutôt avec des chaînes de caractères à la place de noms d'attribut. D'ailleurs, c'est un peu ce que nous venons de faire, dans nos redéfinitions de méthodes accédant aux attributs.

Là encore, si l'intérêt ne saute pas aux yeux, laissez ces fonctions de côté. Vous pourrez les retrouver par la suite .

Les méthodes de conteneur

Nous allons commencer à travailler sur ce que l'on appelle la **surcharge d'opérateurs**. Il s'agit assez simplement d'expliquer à Python quoi faire quand on utilise tel ou tel opérateur. Nous allons ici voir quatre méthodes spéciales qui interviennent quand on travaille sur des objets conteneurs.

Accès aux éléments d'un conteneur

Les objets conteneurs, j'espère que vous vous en souvenez , ce sont les chaînes de caractères, les listes et les dictionnaires entre autre. Tous ont un point commun : ils contiennent d'autres objets, et on peut accéder à ces objets contenus grâce à l'opérateur *[]*.

Les trois premières méthodes que nous allons voir sont *__getitem__*, *__setitem__* et *__delitem__*. Elles servent respectivement à définir :

- Quoi faire quand on entre `objet[index]`
- Quoi faire quand on entre `objet[index] = valeur`
- Quoi faire quand on entre `del objet[index]`

Pour cet exemple, nous allons voir une classe enveloppe de dictionnaire. Les classes enveloppes sont des classes qui ressemblent à d'autres classes, mais n'en sont pas réellement. Ca vous avance ?

Nous allons créer une classe que nous allons appeler *ZDict*. Elle va posséder un attribut auquel on ne devra pas accéder de l'extérieur de la classe, un dictionnaire que nous appelerons *_dictionnaire*. Quand on créera un objet de type *ZDict* et qu'on voudra faire *objet[index]*, à l'intérieur de la classe on fera *self._dictionnaire[index]*. En réalité, notre classe fera semblant d'être un dictionnaire, elle réagira pareil, mais elle n'en sera pas réellement un.

Code : Python

```
class ZDict:
    """Classe enveloppe d'un dictionnaire"""
    def __init__(self):
        """Notre classe n'accepte aucun paramètre"""
        self._dictionnaire = {}
    def __getitem__(self, index):
        """Cette méthode spéciale est appelée quand on fait
objet[index]
Elle redirige vers self._dictionnaire[index]

"""
        return self._dictionnaire[index]
    def __setitem__(self, index, valeur):
        """Cette méthode est appelée quand on entre objet[index] =
valeur
On redirige vers self._dictionnaire[index] = valeur

"""
        self._dictionnaire[index] = valeur
```

Vous avez un exemple d'utilisation des deux méthodes *__getitem__* et *__setitem__* qui, je pense, est assez clair. Pour *__delitem__*, je pense que c'est assez évident, elle ne prend qu'un seul paramètre qui est l'index que l'on souhaite supprimer. Vous pouvez étendre cet exemple avec d'autres méthodes que nous avons vu plus haut, notamment *__repr__* et *__str__*. N'hésitez pas, entraînez-vous, tout ça peut vous servir .

La méthode spéciale derrière le mot-clé in

Il existe une quatrième méthode, appelée *__contains__*, qui est utilisée quand on souhaite savoir si un objet se trouve dans un conteneur.

Exemple classique :

Code : Python

```
ma_liste = [1, 2, 3, 4, 5]
8 in ma_liste # revient au même que ...
ma_liste.__contains__(8)
```

Ainsi, si vous voulez que votre classe enveloppe puisse utiliser le mot-clé *in* comme une liste ou un dictionnaire, vous devez redéfinir cette méthode *__contains__* qui prend en paramètre, outre *self*, l'objet qui nous intéresse. Si l'objet est dans le conteneur, on doit retourner *True*, et *False* sinon.

Je vous laisse redéfinir cette méthode, vous avez toutes les indications nécessaires  .

Connaître la taille d'un conteneur

Il existe enfin une méthode spéciale *__len__*, appelée quand on souhaite connaître la taille d'un objet conteneur, grâce à la fonction *len*.

len(objet) équivaut à *objet.__len__()*. Cette méthode spéciale ne prend aucun paramètre et retourne une taille, sous la forme d'un entier. Là encore, je vous laisse faire l'essai  .

Les méthodes mathématiques

Pour cette sous-partie, nous allons continuer à voir les méthodes spéciales permettant la surcharge d'opérateurs mathématiques, comme +, -, * et j'en passe.

Ce qu'il faut savoir

Pour cette sous-partie, nous allons utiliser un nouvel exemple, une classe capable de contenir des durées. Ces durées seront contenues sous la forme d'un nombre de minutes et un nombre de secondes.

Voici le corps de la classe, gardez-le sous la main :

Code : Python

```
class Duree:
    """Classe contenant des durées sous la forme d'un nombre de
minutes
et de secondes

"""
    def __init__(self, min=0, sec=0):
        """Constructeur de la classe"""
        self.min = min # nombre de minutes
        self.sec = sec # nombre de secondes
    def __str__(self):
        """Affichage un peu plus joli de nos objets"""
        return "{0:02}:{1:02}".format(self.min, self.sec)
```

On définit simplement deux attributs contenant notre nombre de minutes et notre nombre de secondes, ainsi qu'une méthode pour afficher tout cela un peu mieux. Si vous vous interrogez sur l'utilisation de la méthode *format* dans la méthode *__str__*, sachez simplement que le but est de voir la durée sous la forme MM:SS (voir la documentation de Python concernant le formatage des chaînes pour plus d'informations).

Créons un premier objet *Duree* que nous appelons *d1*.

Code : Python Console

```
>>> d1 = Duree(3, 5)
>>> print(d1)
03:05
>>>
```

Si vous essayez de faire *d1 + 4* par exemple, vous allez obtenir une erreur. Python ne sait pas comment additionner un type *Duree* et un *int*. Il ne sait même pas comment ajouter deux durées ! Nous allons donc lui expliquer.

La méthode spéciale à redéfinir est *__add__*. Elle prend en paramètre l'objet que l'on souhaite ajouter. Voici deux lignes de code qui reviennent au même :

Code : Python

```
d1 + 4
d1.__add__(4)
```

Comme vous le voyez, quand vous utilisez le symbole + ainsi, c'est en fait la méthode *__add__* de l'objet *Duree* qui est appelée. Elle prend en paramètre l'objet que l'on souhaite ajouter, peu importe le type de l'objet en question. Et elle doit retourner un objet exploitable, ici il serait plus logique que ce soit une nouvelle durée.

Si vous devez faire différentes actions en fonction du type de l'objet à ajouter, testez le résultat de *type(objet_a_ajouter)*.

Code : Python

```
def __add__(self, objet_a_ajouter):
        """L'objet à ajouter est un entier, le nombre de
secondes"""
        nouvelle_duree = Duree()
        # On va copier self dans l'objet créé pour avoir la même
durée
        nouvelle_duree.min = self.min
        nouvelle_duree.sec = self.sec
        # On ajoute la durée
        nouvelle_duree.sec += objet_a_ajouter
        # Si le nombre de secondes >= 60
        if nouvelle_duree.sec >= 60:
            nouvelle_duree.min += nouvelle_duree.sec // 60
            nouvelle_duree.sec = nouvelle_duree.sec % 60
        # On retourne la nouvelle durée
        return nouvelle_duree
```

Prenez le temps de comprendre le mécanisme et le petit calcul pour vous assurer d'avoir une durée cohérente. D'abord, on crée une nouvelle durée qui est l'équivalent de notre durée contenue dans *self*. On lui ajoute le nombre de secondes à ajouter et on s'assure que le temps est cohérent (le nombre de secondes n'atteint pas 60). Si le temps n'est pas cohérent, on le corrige. On retourne enfin notre nouvel objet modifié. Voici un petit code qui montre comment utiliser notre méthode :

Code : Python Console

```
>>> d1 = Duree(12, 8)
>>> print(d1)
12:08
>>> d2 = d1 + 54 # d1 + 54 secondes
>>> print(d2)
13:02
>>>
```

Pour mieux comprendre, vous pouvez remplacer `d2 = d1 + 54` par `d2 = d1.__add__(54)` , cela revient au même. Ce remplacement n'est que pour bien comprendre le mécanisme. Il va de soi que ces méthodes spéciales ne sont pas à appeler directement depuis l'extérieur de la classe, les opérateurs n'ont pas été inventés pour rien  .

Sachez que sur le même modèle, il existe les méthodes :

- *__sub__* : surcharge de l'opérateur - ;
- *__mul__* : surcharge de l'opérateur * ;
- *__truediv__* : surcharge de l'opérateur / ;
- *__floordiv__* : surcharge de l'opérateur // (division entière) ;
- *__mod__* : surcharge de l'opérateur % (modulo) ;
- *__pow__* : surcharge de l'opérateur ** (puissance) ;
- …

Il y en a d'autres que vous pouvez consulter sur la documentation de votre version de Python, sur le site officiel.

Tout dépend du sens

Vous l'avez peut-être remarqué, et c'est assez logique si vous avez suivi mes explications, mais écrire *objet1 + objet2* ne revient pas au même qu'écrire *objet2 + objet1*, si les deux objets ont des types différents.

En effet, ce sont les méthodes *__add__* de chacun des objets qui sont appelées, dans un cas ou dans l'autre.

Cela signifie que quand on utilise notre classe *Duree*, si nous entrons `d1 + 4` cela marche, alors que `4 + d1` ne marche pas. En effet, la class *int* ne sait pas quoi faire de votre objet *Duree*.

Il existe cependant une panoplie de méthodes spéciales pour faire le travail de *__add__* mais si vous entrez l'opération dans l'autre sens. Il suffit de préfixer le nom des méthodes spéciales par un *r*.

Code : Python

```
def __radd__(self, objet_a_ajouter):
        """Cette méthode est appelée si on entre 4 + objet et que
l'objet à ajouter (4 dans cet exemple) ne sait pas comment ajouter
notre donnée. On se contente de rediriger sur __add__, puisque
ici ça revient au même, l'opération doit avoir le même résultat,
posée dans un sens où dans l'autre.

"""
        return self + objet_a_ajouter
```

À présent, on peut entrer *4 + d1*, ça revient au même que *d1 + 4*.

N'hésitez pas à relire ces exemples s'ils vous paraissent peu clairs.

D'autres opérateurs

Il est également possible de surcharger les opérateurs +=, -=… On préfixe les noms de méthode que nous avons vus par un *i* cette fois.

Exemple de méthode *__iadd__* pour notre classe *Duree* :

Code : Python

```
def __iadd__(self, objet_a_ajouter):
        """L'objet à ajouter est un entier, le nombre de
secondes"""
        # On travaille directement sur self cette fois
        # On ajoute la durée
```

```
            self.sec += objet_a_ajouter
            # Si le nombre de secondes >= 60
            if self.sec >= 60:
                self.min += self.sec // 60
                self.sec = self.sec % 60
            # On retourne self
            return self
```

Et en image :

Code : Python Console

```
>>> d1 = Duree(8, 5)
>>> d1 += 128
>>> print(d1)
10:13
>>>
```

Je ne peux que vous encourager à faire des tests, pour être bien sûr de comprendre le mécanisme. Je vous ai donné ici une façon de faire en la commentant, mais si vous ne pratiquez pas, ou n'essayez pas par vous-même, vous n'allez pas la retenir et vous n'allez pas forcément comprendre la logique.

Les méthodes de comparaison

Pour finir, nous allons voir la surcharge des opérateurs de comparaison que vous connaissez depuis quelque temps maintenant : ==, *!=*, <, >, <=, >=.

Ces méthodes sont donc appelées si vous tentez de comparer deux objets entre eux. Comment Python sait-il que 3 est inférieur à 18 ? Une méthode spéciale de la classe *int* le permet, en simplifiant. Donc si vous voulez comparer des durées par exemple, vous allez devoir redéfinir certaines méthodes que je vais présenter plus bas. Elles devront prendre en paramètre l'objet à comparer à *self*, et doivent retourner un booléen (*True* ou *False*).

Je vais me contenter de vous faire un petit tableau récapitulatif des méthodes à redéfinir pour comparer deux objets entre eux :

Opérateur	Méthode spéciale	Résumé
==	`def __eq__(self, objet_a_comparer):`	Opérateur d'égalité (equal). Retourne *True* si *self* et *objet_a_comparer* sont égaux, *False* sinon
!=	`def __ne__(self, objet_a_comparer):`	Différent de (non equal). Retourne *True* si *self* et *objet_a_comparer* sont différents, *False* sinon
>	`def __gt__(self, objet_a_comparer):`	Test si *self* est strictement supérieur (greater than) à *objet_a_comparer*
>=	`def __ge__(self, objet_a_comparer):`	Test si *self* est supérieur ou égal (greater or equal) à *objet_a_comparer*
<	`def __lt__(self, objet_a_comparer):`	Test si *self* est strictement inférieur (lower than) à *objet_a_comparer*
<=	`def __le__(self, objet_a_comparer):`	Test si *self* est inférieur ou égal (lower or equal) à *objet_a_comparer*

Sachez que ce sont ces méthodes spéciales qui sont appelées si vous voulez trier une liste qui contiendra vos objets, par exemple.

Sachez également que, si Python n'arrive pas à faire *objet1* < *objet2*, il essayera l'opération inverse, soit *objet2* >= *objet1*. Cela est vrai pour les autres opérateurs de comparaison que nous venons de voir.

Aller, je vais vous mettre deux exemples malgré tout, il ne tient qu'à vous de redéfinir les autres méthodes présentées plus haut

:

Secret (cliquez pour afficher)

Code : Python

```
    def __eq__(self, autre_duree):
            """Test si self et autre_duree sont égales"""
            return self.sec == autre_duree.sec and self.min ==
autre_duree.min
        def __gt__(self, autre_duree):
            """Test si self > autre_duree"""
            # On calcule le nombre de secondes de self et autre_duree
            nb_sec1 = self.sec + self.min * 60
            nb_sec2 = autre_duree.sec + autre_duree.min * 60
            return nb_sec1 > nb_sec2
```

Ces exemples devraient vous suffir, je pense .

Des méthodes spéciales utiles à pickle

Vous vous souvenez de *pickle* j'espère . Pour conclure ce chapitre sur les méthodes spéciales, nous allons en voir deux qui sont utilisées par ce module pour influencer la façon dont nos objets sont enregistrés en fichier.

Prenons un cas concret, d'une utilité pratique discutable .

On crée une classe qui va contenir plusieurs attributs. Un de ces attributs possède une valeur temporaire, qui n'est utile que pendant l'exécution du programme. Si on arrête ce programme et qu'on le relance, on doit récupérer le même objet mais la valeur temporaire doit être remise à *0* par exemple.

Il y a d'autres moyens d'y parvenir, je le reconnais avec vous. Mais les autres applications que j'ai en tête sont plus dures à développer et à expliquer rapidement, donc gardons cet exemple .

La méthode spéciale __getstate__

La méthode *__getstate__* est appelée au moment de sérialiser l'objet. Quand vous voulez enregistrer l'objet à l'aide du module *pickle*, *__getstate__* va être appelée juste avant l'enregistrement.

Si aucune méthode *__getstate__* n'est définie, *pickle* enregistre le dictionnaire des attributs de l'objet à enregistrer. Vous vous rappelez ? Il est contenu dans *objet.__dict__*.

Sinon, *pickle* enregistre dans le fichier la valeur retournée par *__getstate__* (généralement, un dictionnaire d'attributs modifié).

Voyons un peu comment coder notre exemple grâce à *__getstate__* :

Code : Python

```
class Temp:
    """Classe contenant plusieurs attributs, dont un temporaire"""

    def __init__(self):
        """Constructeur de notre objet"""
        self.attribut_1 = "une valeur"
        self.attribut_2 = "une autre valeur"
        self.attribut_temporaire = 5

    def __getstate__(self):
        """Retourne le dictionnaire d'attributs à sérialiser"""
        dict_attr = dict(self.__dict__)
        dict_attr["attribut_temporaire"] = 0
        return dict_attr
```

Avant de revenir sur le code, vous pouvez voir les effets. Si vous tentez d'enregistrer cet objet grâce à *pickle* et que vous le récupérez ensuite depuis le fichier, vous constatez que l'attribut *attribut_temporaire* est à *0*, peu importe sa valeur d'origine.

Voyons le code de *__getstate__*. La méthode ne prend aucun argument (excepté *self* puisque c'est une méthode d'instance).

Elle enregistre le dictionnaire des attributs dans une variable locale *dict_attr*. Ce dictionnaire a le même contenu que *self.__dict__* (le dictionnaire des attributs de l'objet). En revanche, il a une référence différente. Sans quoi, à la ligne suivante, au moment de modifier *attribut_temporaire*, le changement ce serai également appliqué à l'objet, ce que l'on veut éviter.

A la ligne suivante, donc, on change la valeur de l'attribut *attribut_temporaire*. Etant donné que *dict_attr* et *self.__dict__* n'ont pas la même référence, l'attribut n'est changé que dans *dict_attr* et le dictionnaire de *self* n'est pas modifié.

Enfin, on retourne *dict_attr*. Cela fait qu'au lieu d'enregistrer dans notre fichier *self.__dict__*, *pickle* enregistre notre dictionnaire modifié, *dict_attr*.

Si ce n'est pas assez clair, je vous encourage à tester par vous-même, essayez de modifier la méthode *__getstate__* et manipulez *self.__dict__* pour bien comprendre le code.

La méthode __setstate__

A la différence de *__getstate__*, la méthode *__setstate__* est appelée au moment de désérialiser l'objet. Concrètement, si vous récupérez un objet à partir d'un fichier sérialisé, *__setstate__* sera appelée après la récupération du dictionnaire des attributs.

Pour schématiser, voici l'exécution que l'on va observer derrière *unpickler.load()* :

1. L'objet *Unpickler* lit le fichier
2. Il récupère le dictionnaire des attributs. Je vous rappelle que si aucune méthode *__getstate__* n'est définie dans notre classe, ce dictionnaire est celui contenu dans l'attribut spécial *__dict__* de l'objet au moment de sa sérialisation
3. Ce dictionnaire récupéré est envoyé à la méthode *__setstate__* si elle existe. Si elle n'existe pas, Python considère que c'est le dictionnaire des attributs de l'objet à récupérer et écrit donc l'attribut *__dict__* de l'objet en y plaçant ce dictionnaire récupéré.

Le même exemple, mais par la méthode *__setstate__* cette fois :

Code : Python

```
...
    def __setstate__(self, dict_attr):
        """Méthode appelée lors de la désérialisation de l'objet"""
        dict_attr["attribut_temporaire"] = 0
        self.__dict__ = dict_attr
```

Quelle est la différence entre les deux méthodes que nous avons vu ?

L'objectif que nous nous étions fixé peut être atteint par ces deux méthodes. Soit notre classe implémente une méthode *__getstate__*, soit elle implémente une méthode *__setstate__*.

Dans le premier cas, on modifie le dictionnaire des attributs avant la sérialisation. Le dictionnaire des attributs enregistré est celui que nous avons modifié avec la valeur de notre attribut temporaire à *0*.

Dans le second cas, on modifie notre dictionnaire d'attributs après la désérialisation. Le dictionnaire que l'on récupère contient un attribut *attribut_temporaire* avec une valeur quelconque (on ne sait pas laquelle) mais avant de récupérer l'objet, on met cette valeur à *0*.

Ce sont deux moyens différents, qui ici reviennent au même. A vous de choisir la meilleure en fonction de vos besoins (les deux peuvent être présentes dans la même classe si nécessaire).

Là encore, je vous encourage à faire des essais si ce n'est pas très clair.

On peut enregistrer en fichier autre chose que des dictionnaires

Votre méthode *__getstate__* n'est pas obligée de retourner un dictionnaire d'attributs. Elle peut retourner un autre objet, un entier, un flottant, mais dans ce cas une méthode *__setstate__* devra exister pour savoir "quoi faire" avec l'objet enregistré. Si ce n'est pas un dictionnaire d'attributs, Python ne peut pas le deviner .

Là encore, je vous laisse tester si ça vous intéresse .

Je veux encore plus puissant !

__getstate__ et *__setstate__* sont les deux méthodes les plus connues pour agir sur la sérialisation d'objets. Mais il en existe d'autres, plus complexes.

Si vous êtes intéressé, jetez un oeil du côté de la PEP 307 -- Extensions to the pickle protocol.
Ouf ! Ce fut long, mais indispensable . Ce chapitre présentait pas mal de méthodes. Vous ne devez pas toutes les retenir, simplement savoir où chercher dans un contexte où vous pensez en avoir besoin.

Pour une liste exhaustive des méthodes spéciales, je vous renvoie à la documentation officielle de votre version de Python. Pour la V3.X, vous pouvez trouver cette liste à l'adresse http://docs.python.org/py3k/reference/ [...] -method-names. N'hésitez pas à y jeter un coup d'œil, ou à lire plus complètement la documentation, voire le tutoriel mis à votre disposition, c'est peut-être en anglais, mais c'est bien plus complet que ce que je pourrais transmettre .

En attendant, nous allons jeter un œil du côté de l'héritage, quelque chose que vous devez absolument connaître !

L'héritage

J'entends souvent dire qu'un langage de programmation orienté objet n'incluant pas l'héritage serait incomplet, sinon inutile. Après avoir découvert par moi-même cette fonctionnalité, et les techniques qui en découlent, je suis forcé de reconnaître que sans l'héritage, le monde serait moins beau .

Qu'a cette fonctionnalité de si utile ?

Nous allons le voir, bien entendu. Et je vais surtout essayer de vous montrer des exemples d'application. Car très souvent, quand on découvre l'héritage, on ne sait pas trop quoi en faire... avant qu'il ne soit trop tard .

Ne vous attendez donc pas à un chapitre où vous n'allez faire que coder. Vous allez devoir vous pencher sur de la théorie et travailler sur quelques exemples de modélisation. Mais je vous guide, n'en doutez pas .

Pour bien commencer

Je ne vais pas faire durer le suspense plus longtemps : l'héritage est une fonctionnalité objet qui permet de déclarer que telle classe sera elle-même modelée sur une autre classe, qu'on appelle la classe parente, ou la **classe-mère**. Concrètement, si une classe *b* **hérite** de la classe *a*, les objets créés sur le modèle de la classe *b* auront accès aux méthodes et attributs de la classe *a*.

Et c'est tout ? Ca ne sert à rien !

Non, ce n'est pas tout, et si, ça sert énormément , mais vous allez devoir me laisser un peu de temps pour vous montrer l'intérêt .

La première chose, c'est que la classe *b* dans notre exemple ne se contente pas seulement de reprendre les méthodes et attributs de la classe *a*. Elle va pouvoir en définir d'autres. D'autres méthodes et d'autres attributs qui lui seront propres, en plus des méthodes et attributs de la classe *a*. Et elle va pouvoir redéfinir les méthodes de la classe-mère, également.

Prenons un petit exemple : vous définissez une classe particulière qui va implémenter, disons, certaines méthodes pour pouvoir être enregistrée dans des fichiers et lu depuis ces fichiers. Vous voulez utiliser cette classe dans une application qui contiendra beaucoup d'autres classes, dont certaines devront être enregistrées dans des fichiers. Le plus simple est de faire hériter de la classe que vous venez de coder toutes les classes qui devront produire des objets destinés à être enregistrés dans des fichiers. Si vous vous rendez compte qu'un bug se produit lors de l'enregistrement, vous n'aurez à modifier que la classe-mère, au lieu de modifier toutes les classes-filles.

Cet exemple est un exemple d'utilisation de l'héritage. On peut le retrouver dans beaucoup de schémas différents cependant.

Autre exemple, un programme gérant des animaux : les naturalistes ont passé des siècles à répertorier les animaux connus et à les regrouper selon une classification rigoureuse, comprenant des règnes, des embranchements, des classes, des groupes... pour finalement arriver aux espèces et aux variétés (j'en oublie, naturellement). La classe des mammifères, par exemple, comprend plusieurs sous-classes, comprenant elles-mêmes plusieurs ordres... vous saisissez l'idée. Chaque point de cette classification possède ainsi certaines caractéristiques propres que l'on retrouve dans les divisions de la classification.

Ici, on pourrait construire une classe *Mammifere*, avec comme attributs particulier "l'allaitement des petits". Ensuite, il faudrait définir d'autres classes, héritées de Mammifere, possédant des attributs propres à cette subdivision des mammifères.

Ces classes seraient aussi amenées à être héritées. Quand on a réussi le tour de force de représenter la classification utilisée par les naturalistes dans une hiérarchie d'objets hérités, il suffit de créer un animal, par exemple un éléphant de savane, qui reprendra les attributs de son règne animal, de sa classe mammifère, de la famille des éléphantidés (il en manque pas mal mais c'est pour l'idée).

Un autre exemple ? On se rapproche de mon terrain de chasse . Imaginons un jeu en ligne, un jeu de rôle pour plus de facilité. Vous n'ignorez sans doute pas que ce type de jeu permet à plusieurs joueurs, se connectant de tous les coins du monde d'incarner un personnage. Ce personnage peut interragir avec les autres joueurs connectés, mais aussi avec l'univers hébergé par le serveur de jeu en ligne. On se retrouve donc avec plusieurs types de personnage :

- Les bots ou NPCs (Non-Playing Character) qui sont des personnages dirigés par une intelligence artificielle. Ils ne

représentent pas des joueurs connectés mais des personnages de l'univers virtuel, dirigés par l'ordinateur
- Les joueurs, comme vous et moi, qui sont connectés au jeu en ligne et interragissent avec l'univers et les autres connecté s
- Les administrateurs, en charge de construire l'univers, de veiller au bon fonctionnement du jeu voire de coder de nouvelles choses .

C'est une liste sommaire, très incomplète des différents objets que nous pourrions avoir. Si vous essayez de trier les choses dans votre esprit (ou sur papier), vous pouvez vous rendre compte que beaucoup de ces personnages reprennent des caractéristiques identiques. Par exemple, les NPCs, tout comme les joueurs et les administrateurs, doivent pouvoir se déplacer dans l'univers, lancer des sorts, combattre... mais les administrateurs auront probablement une puissance infinie et ne pourront pas être tués. De même, l'intelligence artificielle qui se cache derrière les NPCs ne doit pas s'appliquer à des joueurs connectés, puisque ce sont eux qui contrôlent leur personnage.

Pour résumer, tous ces objets ont des points communs et des différences. Nous pourrions donc modéliser notre programme avec une hiérarchie d'héritage ressemblant à celle-ci :

- Une classe *Personnage* qui contiendra les attributs et méthodes communs à tous les personnages que nous avons vu. La capacité de se déplacer, combattre, lancer des sorts, et bien d'autres, seront définies dans cette classe.
  - Une classe *NPC*, héritée de *Personnage*. Elle reprendra les méthodes et attributs de la classe *Personnage* en se dotant de nouveaux attributs et méthodes, comme par exemple un mécanisme d'intelligence artificielle
  - Une classe *Joueur*, héritée de *Personnage*. Tout comme *NPC*, elle reprend les mécanismes de déplacement, de combat définis dans *Personnage* mais elle ajoutera d'autres attributs et méthodes qui seront liés au fait que c'est un joueur connecté. Il n'aura pas besoin d'intelligence artificielle, puisqu'il est contrôlé à la souris et au clavier
    - Une classe *Administrateur* héritée de *Joueur*. Elle reprendra non seulement les attributs et méthodes de *Joueur*, mais aussi ceux de *Personnage* dont *Joueur* est héritée. Cette classe donnera accès à certaines fonctionnalités inaccessibles aux joueurs. On peut imaginer faire hériter cette classe de plusieurs autres qui représenteront différents niveaux d'administrateur.

La hiérarchie que je propose est loin d'être la seule possible. J'ai surtout essayé de vous montrer l'intérêt de l'héritage dans des situations assez diverses et concrètes, pour que vous vous fassiez votre idée. Les mécanismes de l'héritage, que nous allons voir maintenant, ne sont pas bien compliqués. En général, c'est surtout les idées d'application qui manquent au début. Et un programme avec beaucoup de classes sans relation d'héritage entre elles sera, d'une façon générale, moins modulable qu'un programme bien hiérarchisé.

L'héritage simple

On oppose l'**héritage simple** que nous venons de voir théoriquement dans la sous-partie précédente, à l'**héritage multiple** que nous verrons dans la sous-partie suivante.

Il est temps de voir la syntaxe de l'héritage. Nous allons définir une première classe *A* et une seconde classe *B*, héritée de *A*.

Code : Python

```
class A:
    """Classe A, pour illustrer notre exemple d'héritage."""
    pass # on laisse notre définition vide, ce n'est qu'un exemple

class B(A):
    """Classe B, héritée de A.
Elle reprend les mêmes méthodes et attributs (dans cet exemple, la
classe
A ne possède de toute façon ni méthodes ni attributs).

"""
    pass
```

Vous pourrez expérimenter par la suite des exemples plus constructifs . Pour l'instant, l'important est de bien noter la syntaxe,

qui comme vous le voyez est des plus simple : `class MaClasse(MaClasseMere):`. Dans la définition de la classe, entre le nom et les deux points, vous précisez entre parenthèses la classe de laquelle elle doit hériter. Comme je l'ai dit, dans un premier temps toutes les méthodes de la classe *A* se retrouveront dans la classe *B*.

J'ai essayé de mettre des constructeurs dans les deux classes, mais dans la classe héritée je ne retrouve pas les attributs déclarés dans ma classe-mère, c'est normal ?

Tout à fait. Vous vous souvenez quand je vous ai dit que les méthodes étaient définies dans la classe, alors que les attributs étaient directement déclarés dans l'instance d'objet ? Vous le voyez bien de toute façon : c'est dans notre constructeur qu'on déclare nos attributs, et on les écrit tous dans l'instance *self*.

Quand une classe *B* hérite d'une classe *A*, les objets de type *B* reprennent bel et bien les méthodes de la classe *A* en même temps que celles de la classe *B*. Mais, assez logiquement, ce sont celles de la classe *B* qui sont appelées d'abord :

Si vous faites *objet_de_type_b.ma_methode()*, Python va d'abord chercher la méthode *ma_methode* dans la classe *B* dont l'objet est directement issu. Si il ne trouve pas, il va chercher récursivement dans les classes dont hérite *B*, c'est-à-dire *A* dans notre exemple. Ce mécanisme est très important : il induit que si aucune méthode n'a été redéfinie dans la classe, on cherche dans la classe-mère. On peut ainsi redéfinir dans une classe une certaine méthode, et laisser d'autres directement héritées de la classe-mère.

Petit code d'exemple :

Code : Python

```
class Personne:
    """Classe représentant une personne."""
    def __init__(self, nom):
        """Constructeur de notre classe."""
        self.nom = nom
        self.prenom = "Martin"
    def __str__(self):
        """Méthode appelée lors d'une conversion de l'objet en
chaîne"""
        return "{0} {1}".format(self.prenom, self.nom)

class AgentSpecial(Personne):
    """Classe définissant un agent spécial.
Elle hérite de la classe Personne.

"""
    def __init__(self, nom, matricule):
        """Un agent se définit par son nom et son matricule"""
        self.nom = nom
        self.matricule = matricule
    def __str__(self):
        """Méthode appelée lors d'une conversion de l'objet en
chaîne"""
        return "Agent {0}, matricule {1}".format(self.nom,
self.matricule)
```

Vous voyez ici un exemple d'héritage simple. Seulement, si vous essayez de créer des agents spéciaux, vous risquez d'avoir de drôles de surprises :

Code : Python Console

```
>>> agent = AgentSpecial("Fisher", "18327-121")
>>> agent.nom
'Fisher'
>>> print(agent)
Agent Fisher, matricule 18327-121
>>> agent.prenom
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
AttributeError: 'AgentSpecial' object has no attribute 'prenom'
>>>
```

Argh... mais tu avais pas dit qu'une classe reprenait les méthodes et attributs de sa classe-mère ?

Si. Mais en suivant bien l'exécution, vous allez comprendre : tout commence à la création de l'objet. Quel constructeur appeler ? Si il n'y avait pas de constructeur défini dans notre classe *AgentSpecial*, Python appellerait celui de *Personne*. Mais il en existe bel et bien un dans la classe *AgentSpecial* et c'est donc celui-ci qui est appelé. Dans ce constructeur, on définit deux attributs, *nom* et *matricule*. Mais c'est tout : le constructeur de la classe *Personne* n'est pas appelé, sauf si vous l'appelez explicitement dans le constructeur d'*AgentSpecial*.

Dans le premier chapitre, je vous ai expliqué que *mon_objet.ma_methode()* revenait au même que *MaClasse.ma_methode(mon_objet)*. Dans notre méthode *ma_methode*, le premier paramètre *self* sera *mon_objet*. Nous allons nous servir de cette équivalence. La plupart du temps, faire *mon_objet.ma_methode()* suffit. Mais dans une relation d'héritage, il peut y avoir, comme nous l'avons vu, plusieurs méthodes du même nom définies dans différentes classes. Laquelle appeler ? Python choisit, si il la trouve, celle définie directement dans la classe dont est issu l'objet, et sinon parcourt la hiérarchie de l'héritage jusqu'à tomber sur la méthode. Mais on peut aussi se servir de la notation *MaClasse.ma_methode(mon_objet)* pour appeler une méthode précise d'une classe précise. Et cela est utile dans notre cas  :

Code : Python

```
class Personne:
    """Classe représentant une personne."""
    def __init__(self, nom):
        """Constructeur de notre classe."""
        self.nom = nom
        self.prenom = "Martin"
    def __str__(self):
        """Méthode appelée lors d'une conversion de l'objet en
chaîne"""
        return "{0} {1}".format(self.prenom, self.nom)

class AgentSpecial(Personne):
    """Classe définissant un agent spécial.
Elle hérite de la classe Personne.

"""
    def __init__(self, nom, matricule):
        """Un agent se définit par son nom et son matricule"""
        # on appelle explicitement le constructeur de Personne :
        Personne.__init__(self, nom)
        self.matricule = matricule
    def __str__(self):
        """Méthode appelée lors d'une conversion de l'objet en
chaîne"""
        return "Agent {0}, matricule {1}".format(self.nom,
self.matricule)
```

Si ça vous paraît encore un peu vague, expérimentez : c'est toujours le meilleur moyen. Entraînez-vous, contrôlez l'écriture des attributs, ou revenez au premier chapitre de cette partie pour vous rafraîchir la mémoire au sujet du paramètre *self*, bien qu'à force de manipulations vous avez dû comprendre l'idée.

Reprenons notre code de tout à l'heure qui, cette fois, passe sans problème :

Code : Python Console

```
>>> agent = AgentSpecial("Fisher", "18327-121")
>>> agent.nom
```

```
'Fisher'
>>> print(agent)
Agent Fisher, matricule 18327-121
>>> agent.prenom
'Martin'
>>>
```

Cette fois, notre attribut *prenom* se trouve bien dans notre agent spécial, car le constructeur de la classe *AgentSpecial* appelle explicitement celui de *Personne*.

Vous pouvez noter également que dans notre constructeur d'*AgentSpecial*, on n'instancie pas notre attribut *nom*. Celui-ci est en effet écrit par le constructeur de la classe *Personne* que nous appelons en lui passant en paramètre le nom de notre agent.

Petite précision

Dans le chapitre précédent, je suis passé très rapidement sur l'héritage, ne voulant pas trop m'y attarder et brouiller les cartes inutilement  . Mais j'ai expliqué brièvement que toutes les classes que vous créez héritent de la classe *object*. C'est elle, notamment, qui définit toutes les méthodes spéciales que nous avons vu dans le chapitre précédent et qui connaît, bien mieux que nous, le mécanisme interne de l'objet. Vous devriez un peu mieux, à présent, comprendre le code du chapitre précédent. Le voici, en substance :

Code : Python

```
def __setattr__(self, nom_attribut, valeur_attribut):
        """Méthode appelée quand on fait objet.attribut = valeur"""
        print("Attention, on modifie l'attribut {0} de l'objet
!".format(nom_attribut))
        object.__setattr__(self, nom_attribut, valeur_attribut)
```

En redéfinissant notre méthode *__setattr__*, on ne peut, dans le corps de cette méthode, modifier les valeurs de nos attributs comme on le fait habituellement (*self.attribut = valeur*) car alors, la méthode s'appellerait elle-même. Donc on fait appel à la méthode *__setattr__* de la classe *object*, cette classe dont héritent implicitement toutes nos classes. On est sûr que la méthode de cette classe sait écrire une valeur dans un attribut, alors que nous ignorons le mécanisme et que nous n'avons pas besoin de le connaître : c'est la magie du procédé, une fois qu'on a bien compris le principe  .

Deux fonctions très pratiques

Python définit deux fonctions qui peuvent s'avérer utiles dans bien des cas : *issubclass* et *isinstance*.

issubclass

Comme son nom l'indique, elle vérifie si une classe est une sous-classe d'une autre classe. Elle retourne *True* si c'est le cas, *False* sinon :

Code : Python Console

```
>>> issubclass(AgentSpecial, Personne) # AgentSpecial hérite de
Personne
True
>>> issubclass(AgentSpecial, object)
True
>>> issubclass(Personne, object)
True
>>> issubclass(Personne, AgentSpecial) # Personne n'hérite pas
d'AgentSpecial
False
>>>
```

isinstance

isinstance permet de savoir si un objet est issu d'une classe ou de ses classe-filles :

Code : Python

```
>>> agent = AgentSpecial("Fisher", "18327-121")
>>> isinstance(agent, AgentSpecial) # agent est une instance
d'AgentSpecial
True
>>> isinstance(agent, Personne) # agent est une instance héritée de
Personne
True
>>>
```

Ces quelques exemples suffisent, je pense. Peut-être devrez-vous attendre un peu avant de trouver une utilité à ces deux fonctions, mais ce moment viendra  .

L'héritage multiple

Python inclut un mécanisme permettant l'**héritage multiple**. L'idée est en substance très simple : au lieu d'hériter d'une seule classe, on peut en hériter de plusieurs.

 Ce n'est pas ce qui se passe, quand on hérite d'une classe qui hérite elle-même d'une autre classe ?

Pas tout à fait. La hiérarchie de l'héritage simple permet d'étendre des méthodes et attributs d'une classe à plusieurs autres, mais la structure reste fermée. Pour mieux comprendre, considérez cet exemple :

On peut s'asseoir dans un fauteuil. On peut dormir dans un lit. Mais on peut s'asseoir et dormir dans certains canapés (la plupart en fait, avec un peu de bonne volonté ). Notre classe *Fauteuil* pourra hériter de la classe *ObjetPourSAsseoir* et notre classe *Lit*, de notre classe *ObjetPourDormir*. Mais notre classe *Canape* alors ? Elle devra logiquement hériter de nos deux classes *ObjetPourSAsseoir* et *ObjetPourDormir*. C'est un cas où l'héritage multiple pourrait s'avérer utile.

Assez souvent, on utilisera l'héritage multiple pour des classes qui ont besoin de certaines fonctionnalités, définies dans une classe-mère. Par exemple, une classe peut produire des objets destinés à être enregistrés dans des fichiers. On peut faire hériter de cette classe toutes celles qui produiront des objets à enregistrer dans des fichiers. Mais ces mêmes classes pourront hériter d'autres classes, incluant, pourquoi pas, d'autres fonctionnalités.

C'est une des utilisations de l'héritage multiple, et il en existe d'autres. Bien souvent, l'utilisation de cette fonctionnalité ne vous semblera évidente qu'en vous penchant sur la hiérarchie d'héritage de votre programme. Pour l'instant, je vais me contenter de vous donner la syntaxe et un peu de théorie supplémentaire, en vous encourageant à essayer par vous-même :

Code : Python

```
class MaClasseHeritee(MaClasseMere1, MaClasseMere2):
```

Vous pouvez faire hériter votre classe de plus de deux autres classes. Au lieu de préciser, comme dans les cas d'héritage simple, une seule classe-mère entre parenthèses, vous en précisez plusieurs, séparées par des virgules.

Recherche des méthodes

La recherche des méthodes se fait dans l'ordre de la définition de la classe. Dans l'exemple ci-dessus, si on appelle une méthode d'un objet issu de *MaClasseHeritee*, on va d'abord chercher dans la classe *MaClasseHeritee*. Si la méthode n'est pas trouvée, on la cherche d'abord dans *MaClasseMere1*. Encore une fois, si la méthode n'est pas trouvée, on cherche dans toutes les classe-mères de la classe *MaClasseMere1*, si elle en a, et selon le même système. Si, encore et toujours, on ne trouve pas la méthode, on

la recherche dans *MaClasseMere2* et ses classe-mères successives, ainsi de suite.

C'est donc l'ordre de définition des classe-mères qui importe. On va chercher la méthode dans les classe-mères de gauche à droite. Dans chaque classe dans laquelle on recherche, si la méthode n'est pas trouvée, on va chercher dans ses classe-mères, et ainsi de suite.

Retour sur les exceptions

Nous ne sommes pas revenu depuis la première partie sur les exceptions. Toutefois, ce chapitre me donne une opportunité d'aller un peu plus loin .

Les exceptions sont non seulement des classes, mais des classes hiérarchisées selon une relation d'héritage précise.

Cette relation d'héritage devient importante quand vous utilisez le mot-clé *except*. En effet, le type de l'exception que vous précisez après est intercepté... ainsi que toutes les classes qui héritent de ce type.

Mais comment fait-on pour savoir qu'une exception hérite d'autres exceptions ?

Il y a plusieurs possibilités. Si vous vous intéressez à une exception en particulier, consultez l'aide qui est liée.

Code : Console

```
Help on class AttributeError in module builtins:

class AttributeError(Exception)
 |  Attribute not found.
 |
 |  Method resolution order:
 |      AttributeError
 |      Exception
 |      BaseException
 |      object
```

Vous apprenez ici que l'exception *AttributeError* hérite de *Exception*, qui hérite elle-même de *BaseException*.

Vous pouvez également retrouver la hiérarchie des exceptions built-in à l'adresse http://docs.python.org/py3k/library/ex[...]ion-hierarchy.

Ne sont répertoriées ici que les exceptions dites built-in. D'autres peuvent être définies dans des modules que vous utiliserez, et vous pouvez même en créer vous-même (nous allons voir ça un peu plus bas).

Pour l'instant, souvenez-vous que quand vous entrez *except TypeException*, vous pourrez intercepter toutes les exceptions du type *TypeException* mais aussi celles des classes héritées de *TypeException*.

La plupart des exceptions sont levées pour signaler une erreur... mais pas toutes. L'exception *KeyboardInterupt* est levée quand vous interrompez votre programme, par exemple par CTRL + C. Si bien que quand on souhaite intercepter toutes les erreurs potentielles, on évitera de rentrer un simple *except:* et on le remplacera par *except Exception:*, toutes les exceptions "d'erreurs" étant dérivées de *Exception*.

Création d'exceptions personnalisées

Il peut vous être utile de créer vos propres exceptions. Puisque les exceptions sont des classes, comme nous venons de le voir, rien ne vous empêche de créer les vôtre. Vous pourrez les lever avec *raise*, les intercepter avec *except*.

Se positionner dans la hiérarchie

Vos exceptions doivent hériter d'une exception built-in proposée par Python. Commencez par parcourir la hiérarchie des exceptions built-in pour voir si votre exception peut être dérivée d'une exception qui lui serai proche. La plupart du temps, vous devrez choisir entre ces deux exceptions :

- *BaseException* : la classe-mère de toutes les exceptions. La plupart du temps si vous faites hériter votre classe de *BaseException*, ce sera pour modéliser une exception qui ne sera pas foncièrement une erreur, par exemple une interruption dans le traitement de votre programme
- *Exception* : c'est de cette classe que vos exceptions hériteront la plupart du temps. C'est la classe-mère de toutes les exceptions "d'erreurs"

Si vous pouvez trouver, dans le contexte, une exception qui se trouve plus bas dans la hiérarchie, c'est toujours mieux.

Que doit contenir notre classe exception ?

Deux choses : un constructeur, et une méthode *__str__* car au moment où l'exception est levée, elle doit être affichée. Souvent, votre constructeur ne prendra en paramètre que le message d'erreur, et la méthode *__str__* retournera ce message :

Code : Python

```
class MonException(Exception):
    """Exception levée dans un certain contexte... qui reste à définir"""
    def __init__(self, message):
        """On se contente de stocker le message d'erreur"""
        self.message = message
    def __str__(self):
        """On retourne le message"""
        return self.message
```

Cette exception s'utilisera le plus simplement du monde :

Code : Python Console

```
>>> raise MonException("OUPS... j'ai tout cassé")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.MonException: OUPS... j'ai tout cassé
>>>
```

Mais vos exceptions peuvent aussi prendre plusieurs paramètres à l'instanciation :

Code : Python

```
class ErreurAnalyseFichier(Exception):
    """Cette exception est levée quand un fichier (de configuration)
n'a pas pu être analysé.

Attributs :
fichier -- le nom du fichier posant problème
ligne -- le numéro de la ligne posant problème
message -- le problème proprement dit

"""
    def __init__(self, fichier, ligne, message):
        """Constructeur de notre exception"""
        self.fichier = fichier
        self.ligne = ligne
        self.message = message
    def __str__(self):
```

```
        """Affichage de l'exception"""
        return "[{0}:{1}]: {2}".format(self.fichier, self.ligne, \
                self.message)
```

Et pour lever cette exception :

Code : Python Console

```
>>> raise ErreurAnalyseFichier("plop.conf", 34, \
...         "il manque une parenthèse à la fin de l'expression")
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
__main__.ErreurAnalyseFichier: [plop.conf:34]: il manque une
parenthèse à la fin de l'expression
>>>
```

Voilà, ce petit retour sur les exceptions est achevé. Si vous voulez en savoir plus, n'hésitez pas à consulter la partie du tutoriel Python officiel consacrée aux exceptions et celle consacrée aux exceptions personnalisées.
Notre tour d'horizon de l'héritage est achevé. C'est un chapitre un peu particulier car on a bien plus travaillé sur des exemples de modélisation que sur du code Python. C'est toutefois un point essentiel et je vous invite à bien y réfléchir avant de vous lancer dans un programme d'une certaine taille  . C'est encore plus vrai ici que dans les chapitres précédents : c'est en pratiquant que vous pourrez bien comprendre l'utilité et maîtriser le concept.

Dans le chapitre suivant, nous allons revenir sur le parcours d'objet, jeter un oeil du côté des itérateurs et des générateurs. Encore un effort et vous allez vous jeter droit dans le TP  .

Derrière la boucle for

Voilà pas mal de chapitres, nous avons étudié les boucles. Ne vous alarmez pas, ce que nous avons vu est toujours d'actualité ... mais nous allons un peu approfondir, maintenant que nous explorons le monde de l'objet .

Nous allons découvrir ces concepts du plus simple au plus complexe, et de telle sorte que chacun des concepts abordés reprenne les précédents. N'hésitez pas, par la suite, à revenir sur ce chapitre et à le relire, partiellement ou intégralement si nécessaire .

Les itérateurs

Nous utilisons des itérateurs sans le savoir depuis le moment où nous avons abordé les boucles et surtout depuis que nous utilisons le mot-clé *for* pour parcourir des objets conteneurs.

Code : Python

```
ma_liste = [1, 2, 3]
for element in ma_liste:
```

Utiliser les itérateurs

C'est sur la seconde ligne que nous allons nous attarder : à force d'utiliser ce type de syntaxe, vous avez dû vous y habituer et ce type de parcours doit vous être familier. Mais il se cache bel et bien un mécanisme derrière cette instruction.

Quand Python tombe sur une ligne du type `for element in ma_liste:` , il va appeler l'itérateur de *ma_liste*. L'itérateur, c'est un objet qui va être chargé de parcourir l'objet conteneur, une liste ici.

L'itérateur est créé dans la méthode spéciale *__iter__* de l'objet. Ici, c'est donc la méthode *__iter__* de la classe *list* qui est appelée et qui retourne un itérateur permettant de parcourir la liste.

À chaque tour de boucle, Python appelle la méthode spéciale *__next__* de l'itérateur qui doit retourner l'élément suivant du parcours ou lever l'exception *StopIteration* si le parcours touche à sa fin.

Ce n'est peut-être pas très clair… alors voyons un exemple.

Sachez avant de plonger dans le code que Python utilise deux fonctions pour appeler et manipuler les itérateurs : *iter* permet d'appeler la méthode spéciale *__iter__* de l'objet passé en paramètre et *next* appelle la méthode spéciale *__next__* de l'itérateur passé en paramètre.

Code : Python Console

```
>>> ma_chaine = "test"
>>> iterateur_de_ma_chaine = iter(ma_chaine)
>>> iterateur_de_ma_chaine
<str_iterator object at 0x00B408F0>
>>> next(iterateur_de_ma_chaine)
't'
>>> next(iterateur_de_ma_chaine)
'e'
>>> next(iterateur_de_ma_chaine)
's'
>>> next(iterateur_de_ma_chaine)
't'
>>> next(iterateur_de_ma_chaine)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

— On commence par créer une chaîne de caractère (jusque là, rien de compliqué).
— On appelle ensuite la fonction *iter* en lui passant en paramètre notre chaîne. Cette fonction appelle la méthode spéciale *__iter__* de notre chaîne qui retourne l'itérateur permettant de parcourir *ma_chaine*.
— On va ensuite appeler plusieurs fois la fonction *next* en lui passant en paramètre notre itérateur. Cette fonction appelle la méthode spéciale *__next__* de notre itérateur. Elle retourne successivement chaque lettre contenue dans notre chaîne et lève une exception *StopIteration* quand la chaîne a été parcourue entièrement.

Quand on parcourt une chaîne grâce à une boucle *for* (`for lettre in chaine:`), c'est ce mécanisme d'itérateur qui est appelé. Chaque lettre retournée par notre itérateur se retrouve dans la variable *lettre* et la boucle s'arrête quand l'exception *StopIteration* est levée.

Vous pouvez reprendre ce code avec d'autres objets conteneurs, des listes par exemple.

Créons nos itérateurs

Pour notre exemple, nous allons créer deux classes :

— *RevStr* : une classe héritée de *str* qui se contentra de redéfinir la méthode *__iter__*. Son mode de parcours sera ainsi altéré : au lieu de parcourir la chaîne de gauche à droite, on la parcourera de droite à gauche (de la dernière lettre à la première).
— *ItRevStr* : notre itérateur. Il sera créé depuis la méthode *__iter__* de *RevStr* et devra parcourir notre chaîne du dernier caractère au premier.

Ce mécanisme est un peu nouveau, je vous mets le code sans trop de suspense. Si vous vous sentez de faire l'exercice, n'hésitez pas, mais je vous donnerai l'occasion de pratiquer dès le prochain chapitre, ne vous inquiétez pas .

Code : Python

```
class RevStr(str):
    """Classe reprenant les méthodes et attributs des chaînes
construites
depuis 'str'. On se contente de définir une méthode de parcours
différente : au lieu de parcourir notre chaîne de la première à la
dernière
lettre, on la parcourt de la dernière à la première.

Les autres méthodes, y compris le constructeur, n'ont pas besoin
d'être redéfinies.

"""
    def __iter__(self):
        """Cette méthode retourne un itérateur parcourant la chaîne
dans le sens inverse de celui de 'str'.

"""
        return ItRevStr(self) # on retourne l'itérateur créé pour
l'occasion

class ItRevStr:
    """Un itérateur permettant de parcourir une chaîne de la
dernière lettre
à la première. On stocke dans des attributs la position courante et
la
chaîne à parcourir.

"""
    def __init__(self, chaine_a_parcourir):
        """On se positionne à la fin de la chaîne"""
        self.chaine_a_parcourir = chaine_a_parcourir
        self.position = len(chaine_a_parcourir)
    def __next__(self):
        """Cette méthode doit retourner l'élément suivant dans le
parcours,
ou lever l'exception 'StopIteration' si le parcours est fini.
```

```
        """
                if self.position == 0: # fin du parcours
                    raise StopIteration
                self.position -= 1 # on décrémente la position
                return self.chaine_a_parcourir[self.position]
```

À présent, vous pouvez créer des chaînes devant se parcourir du dernier caractère vers le premier.

Code : Python Console

```
>>> ma_chaine = RevStr("Bonjour")
>>> ma_chaine
'Bonjour'
>>> for lettre in ma_chaine:
...     print(lettre)
...
r
u
o
j
n
o
B
>>>
```

Sachez qu'il est aussi possible d'implémenter directement la méthode *__next__* dans notre objet conteneur. Dans ce cas, la méthode *__iter__* pourra retourner *self*. Vous pouvez voir un exemple, dont le code ci-dessus est inspiré, dans le tutoriel officiel consacré aux itérateurs.

Ça reste quand même plutôt lourd non, de devoir faire des itérateurs à chaque fois ? Surtout si nos objets conteneurs doivent se parcourir de plusieurs façons, comme les dictionnaires par exemple.

Oui, il subsiste quand même beaucoup de répétitions dans le code que nous devons produire, surtout si nous devons faire plusieurs itérateurs pour un même objet. Souvent, on utilisera des itérateurs existants, par exemple celui des listes. Mais il existe aussi un autre mécanisme, plus simple et plus intuitif : la raison pour laquelle je ne vous montre pas cette autre façon de faire en premier, c'est que cette autre façon passe quand même par des itérateurs, même si c'est implicite, et qu'il n'est pas mauvais de savoir comment cela marche en coulisse .

Il est temps à présent de jeter un oeil du côté des générateurs .

Les générateurs

Les générateurs sont avant tout un moyen plus pratique de créer et manipuler des itérateurs. Vous verrez un peu plus loin dans ce chapitre qu'ils permettent des choses assez complexes, mais leur puissance tient surtout en leur simplicité et leur petite taille .

Les générateurs simples

Pour créer des générateurs, nous allons découvrir un nouveau mot-clé : *yield*. Ce mot-clé ne peut s'utiliser que dans le corps d'une fonction et est suivi d'une valeur à retourner.

Attends un peu… une valeur ? à retourner ?

Oui. Le principe des générateurs étant un peu particulier, il nécessite un mot-clé pour lui tout seul. L'idée est de ne définir une fonction pour un type de parcours. Quand on demande le premier élément du parcours (grâce à *next*), la fonction commence son exécution. Dès qu'elle rencontre une instruction *yield*, elle retourne la valeur qui suit et se met en pause. Quand on demande l'élément suivant de l'objet (grâce, une nouvelle fois, à *next*), l'exécution reprend à l'endroit où elle s'était arrêtée et s'interrompt au

yield suivant… ainsi de suite. À la fin de l'exécution de la fonction, l'exception *StopIteration* est automatiquement levée par Python.

Nous allons prendre un exemple très simple pour commencer :

Code : Python Console

```
>>> def mon_generateur():
...     """Notre premier générateur. Il va simplement retourner 1, 2
et 3"""
...     yield 1
...     yield 2
...     yield 3
...
>>> mon_generateur
<function mon_generateur at 0x00B494F8>
>>> mon_generateur()
<generator object mon_generateur at 0x00B9DC88>
>>> mon_iterateur = iter(mon_generateur())
>>> next(mon_iterateur)
1
>>> next(mon_iterateur)
2
>>> next(mon_iterateur)
3
>>> next(mon_iterateur)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

Je pense que ça vous rappelle quelque chose  . Notre fonction, à part l'utilisation de *yield*, est plutôt classique. Quand on l'exécute, on se retrouve avec un générateur. Ce générateur est un objet créé par Python qui définit sa propre méthode spéciale *__iter__* et donc son propre itérateur. Nous aurions tout aussi bien pu faire :

Code : Python

```
for nombre in mon_generateur(): # attention on exécute la fonction
    print(nombre)
```

Cela rend quand même le code bien plus simple à comprendre.

Notez qu'on doit exécuter notre fonction *mon_generateur* pour obtenir un générateur. Si vous essayez de parcourir notre fonction (`for nombre in mon_generateur`), ça ne marchera pas  .

Bien entendu, la plupart du temps, on ne se contentera pas d'appeler *yield* comme ceci. Notre générateur d'exemples n'a pas beaucoup d'intérêt, il faut bien le reconnaître.

Essayons de faire une chose un peu plus utile : un générateur prenant en paramètres deux entiers, une borne inférieure et une borne supérieure, et retournant chaque entier compris entre ces bornes. Si on entre par exemple *intervalle(5, 25)*, on pourra parcourir les entiers de 6 à 24.

Le résultat attendu est donc :

Code : Python Console

```
>>> for nombre in intervalle(5, 10):
...     print(nombre)
...
6
```

```
7
8
9
>>>
```

Vous pouvez essayer de faire l'exercice, c'est un bon entraînement et pas très compliqué de surcroît.

Au cas où, voici la correction :

Secret (cliquez pour afficher)

Code : Python

```
def intervalle(borne_inf, borne_sup):
    """Générateur parcourant la série des entiers entre borne_inf
et borne_sup.

Note: borne_inf doit être inférieure à borne_sup

"""
    borne_inf += 1
    while borne_inf < borne_sup:
        yield borne_inf
        borne_inf += 1
```

Là encore, vous pouvez améliorer cette fonction. Pourquoi ne pas faire en sorte que si la borne inférieure est supérieure à la borne supérieure, le parcours se fasse dans l'autre sens ?

L'important est que vous compreniez bien l'intérêt et le mécanisme derrière. Je vous encourage, là encore, à tester, à disséquer cette fonctionnalité, à essayer de reprendre les exemples d'itérateurs et à les convertir en générateurs.

Si dans une classe quelconque la méthode spéciale *__iter__* contient un appel à *yield*, alors ce sera ce générateur qui sera appelé quand on voudra parcourir la boucle. Même quand Python passe par des générateurs, comme vous l'avez vu, il utilise (implicitement) des itérateurs. C'est juste plus confortable pour le codeur, on n'a pas besoin de créer une classe par itérateur ni de coder une méthode *__next__*, ni même de lever l'exception *StopIteration*, Python fait tout ça pour nous. Pratique non ?

Les générateurs comme co-routines

Jusqu'ici, que ce soit avec les itérateurs ou les générateurs, nous créons un moyen de parcourir notre objet au début de la boucle *for*, en sachant que nous ne pourrons pas modifier le comportement du parcours par la suite. Mais les générateurs possèdent un certain nombre de méthodes permettant, justement, d'interragir avec eux pendant le parcours.

Malheureusement à notre niveau, les idées d'application utiles me manquent et je vais me contenter de vous montrer la syntaxe et un petit exemple. Peut-être trouverez-vous par la suite une application utile des **co-routines** quand vous vous lancerez dans des programmes conséquents, ou que vous aurez été plus loin dans l'apprentissage du Python  .

Les **co-routines** sont un moyen d'altérer le parcours... pendant le parcours. Par exemple, on pourrait vouloir que dans notre générateur *intervalle*, on passe directement de 5 à 10.

Le système des co-routines en Python est contenu dans le mot-clé *yield* que nous avons vu plus haut et l'utilisation de certaines méthodes de notre générateur.

Interrompre la boucle

La première méthode que nous allons voir est *close*. Elle permet d'interrompre prématurément la boucle, comme le mot-clé *break* en somme.

Code : Python

```
generateur = intervalle(5, 20)
for nombre in generateur:
    if nombre > 17:
        generateur.close() # interruption de la boucle
```

Comme vous le voyez, pour appeler les méthodes du générateur, on doit le stocker dans une variable avant la boucle. Si vous aviez fait directement *for nombre in intervalle(5, 20)*, vous n'auriez pas pu appeler la méthode *close* du générateur.

Envoyer des données à notre générateur

Pour cet exemple, nous allons étendre notre générateur pour qu'il accepte de recevoir des données pendant son exécution.

Le point d'échange de données se fait au mot-clé *yield*. `yield valeur` "retourne" *valeur* qui deviendra donc la valeur courante du parcours. La fonction se met ensuite en pause. On peut, à cet instant, envoyer une valeur à notre générateur. Cela permet d'altérer le fonctionnement de notre générateur pendant le parcours.

Reprenons notre exemple en intégrant cette fonctionnalité :

Code : Python

```
def intervalle(borne_inf, borne_sup):
    """Générateur parcourant la série des entiers entre borne_inf
et borne_sup.
Notre générateur doit pouvoir "sauter" une certaine plage de
nombres
en fonction d'une valeur qu'on lui donne pendant le parcours. La
valeur qu'on lui passe est la nouvelle valeur de borne_inf.

Note: borne_inf doit être inférieure à borne_sup

"""
    borne_inf += 1
    while borne_inf < borne_sup:
        valeur_recue = (yield borne_inf)
        if valeur_recue is not None: # notre générateur a reçu
quelque chose
            borne_inf = valeur_recue
        borne_inf += 1
```

Nous configurons notre générateur pour qu'il accepte une valeur éventuelle au cours du parcours. Si il reçoit une valeur, il va l'attribuer au point du parcours.

Autrement dit, au cours de la boucle, vous pouvez demander que si le nombre est à 15, le générateur saute tout de suite à 20.

Tout se passe sur, et au-dessous, de la ligne du *yield*. Au lieu de simplement renvoyer une valeur à notre boucle, on capture une éventuelle valeur dans *valeur_recue*. La syntaxe est simple : `variable = (yield valeur_a_retourner)` (n'oubliez pas les parenthèses autour de *yield valeur*).

Si aucune valeur n'a été passée à notre générateur, notre *valeur_recue* vaudra *None*. On vérifie donc si elle ne vaut pas *None* et, dans ce cas, on attribue la nouvelle valeur à *borne_inf*.

Voici le code permettant d'interragir avec notre générateur. On utilise la méthode *send* pour envoyer une valeur à notre générateur :

Code : Python Console

```
generateur = intervalle(10, 25)
for nombre in generateur:
if nombre == 15: # on saute à 20
generateur.send(20)
```

```
        print(nombre, end=" ")
```

Il existe d'autres méthodes permettant d'interragir avec notre générateur. Vous pouvez les retrouver, ainsi que des explications supplémentaires, sur la documentation officielle traitant du mot-clé yield.

Voir aussi

- PEP 255 -- simple Generators
- PEP 342 -- Coroutines via Enhanced Generators

Vous savez mieux à présent comment définir des méthodes de parcours pour nos objets conteneurs. Et je dirai que ça tombe à point nommé pour le chapitre suivant... qui est un TP, un TP un peu différent dans son énoncé, mais qui va vous faire pratiquer beaucoup de choses que nous venons de voir dans cette partie  . En avant !

TP 3 : un dictionnaire ordonné

Enfin le temps de la pratique. Vous avez appris pas mal de choses dans la partie courante, beaucoup de concepts, souvent théoriques. Il est temps de les mettre en application, dans un contexte un peu différent des TP précédents : on ne va pas créer un jeu mais plutôt un objet conteneur tenant à la fois du dictionnaire et de la liste.

Notre mission

Notre énoncé va être un peu différent de ceux dont vous avez l'habitude. On ne va pas ici créer un jeu, simplement une classe, destinée à produire des objets conteneurs, des dictionnaires ordonnés.

Peut-être ne vous souvenez-vous pas, je vous ai dit dans le chapitre consacré aux dictionnaires que c'était un type non-ordonné. L'ordre dans lequel vous entrez les données n'a pas d'importance, par exemple. On ne peut ni les trier, ni les inverser, tout cela n'aurai aucun sens pour ce type particulier.

Mais nous allons profiter de l'occasion pour créer une forme de dictionnaire ordonné. L'idée, assez simplement, est de stocker nos données dans deux listes :

- La première contenant nos clés
- La seconde contenant les valeurs correspondantes.

L'ordre d'ajout sera ainsi important, on pourra trier et inverser ce type de dictionnaires.

Spécifications

Voici la liste de ce que notre classe devra implémenter comme mécanisme. Un peu plus bas, vous trouverez un exemple de manipulation de l'objet qui reprend ces spécifications :

1. On doit pouvoir créer notre dictionnaire de plusieurs façons :
   - Vide : on appelle le constructeur sans lui passer aucun paramètre et le dictionnaire créé sera donc vide
   - Copié depuis un dictionnaire : on passe en paramètre du constructeur un dictionnaire que l'on copie dans notre objet. On peut ainsi entrer *constructeur(dictionnaire)* et les clés et valeurs contenues dans le dictionnaire sont copiées dans l'objet construit
   - Pré-rempli grâce à des clés et valeurs passées en paramètre : comme les dictionnaires usuels, on doit pouvoir pré-remplir notre objet avec des couples clés-valeurs passés en paramètre (*constructeur(cle1 = valeur1, cle2 = valeur2, ...)*).
2. Les clés et valeurs doivent être couplées. Autrement dit, si on cherche à supprimer une clé, la valeur correspondante doit être également supprimée. Les clés et valeurs se trouvant dans des listes de même taille, il suffira de prendre l'indice dans une liste pour savoir quel objet correspond dans l'autre. Par exemple, la clé d'indice *0* est couplée avec la valeur d'indice *0*
3. On doit pouvoir interragir avec notre objet conteneur grâce aux crochets, pour récupérer une valeur (*objet[cle]*), ou pour la modifier (*objet[cle] = valeur*) ou pour la supprimer (*del objet[cle]*)
4. Quand on cherche à modifier une valeur, si la clé existe on écrase l'ancienne valeur, si elle n'existe pas on ajoute le couple clé-valeur à la fin du dictionnaire
5. On doit pouvoir savoir grâce au mot-clé *in* si une clé se trouve dans notre dictionnaire (*cle in dictionnaire*)
6. On doit pouvoir demander la taille du dictionnaire grâce à la fonction *len*
7. On doit pouvoir afficher notre dictionnaire, directement dans l'interpréteur ou grâce à la fonction *print*. L'affichage doit être similaire à celui des dictionnaires usuels (*{cle1: valeur1, cle2: valeur2, ...}*)
8. L'objet doit définir les méthodes *sort* pour le trier et *reverse* pour l'inverser. Le tri de l'objet doit se faire en fonction des clé s
9. L'objet doit pouvoir être parcouru. Quand on entre *for cle in dictionnaire*, on doit parcourir la liste des clés contenues dans le dictionnaire
10. A l'instar des dictionnaires, trois méthodes *keys()* (retournant la liste des clés), *values()* (retournant la liste des valeurs) et *items()* (retournant les couples (clé, valeur)) doivent être implémentées. Le type de retour de ces méthodes est laissé à votre initiative : ce peut être des itérateurs ou des générateurs (tant qu'on peut les parcourir)
11. On doit pouvoir ajouter deux dictionnaires ordonnés (*dico1 + dico2*). les clés et valeurs du second dictionnaire sont ajoutées au premier.

Ca vous en fait, du boulot  !

Et vous pourrez encore trouver le moyen d'améliorer votre classe par la suite, si vous le désirez.

Exemple de manipulation

Ci-dessous se trouve un exemple de manipulation de notre dictionnaire ordonné. Quand vous aurez codé le vôtre, vous pourrez ainsi voir si il réagit de la même façon que le mien.

Code : Python Console

```
>>> fruits = DictionnaireOrdonne()
>>> fruits
{}
>>> fruits["pomme"] = 52
>>> fruits["poire"] = 34
>>> fruits["prune"] = 128
>>> fruits["melon"] = 15
>>> fruits
{'pomme': 52, 'poire': 34, 'prune': 128, 'melon': 15}
>>> fruits.sort()
>>> print(fruits)
{'melon': 15, 'poire': 34, 'pomme': 52, 'prune': 128}
>>> legumes = DictionnaireOrdonne(carotte = 26, haricot = 48)
>>> print(legumes)
{'carotte': 26, 'haricot': 48}
>>> len(legumes)
2
>>> legumes.reverse()
>>> fruits = fruits + legumes
>>> fruits
{'melon': 15, 'poire': 34, 'pomme': 52, 'prune': 128, 'haricot': 48,
'carotte':
26}
>>> del fruits['haricot']
>>> 'haricot' in fruits
False
>>> legumes['haricot']
48
>>> for cle in legumes:
...     print(cle)
...
haricot
carotte
>>> legumes.keys()
['haricot', 'carotte']
>>> legumes.values()
[48, 26]
>>> for nom, qtt in legumes.items():
...     print("{0} ({1})".format(nom, qtt))
...
haricot (48)
carotte (26)
>>>
```

Tous au départ !

Je vous ai donné le nécessaire, c'est maintenant à vous de jouer. Concernant l'implémentation, les fonctionnalités, il reste des zones obscures, c'est volontaire. Tout ce qui n'est pas clairement dit est à votre initiative. Tant que ça marche et que l'exemple de manipulation ci-dessus affiche la même chose chez vous, c'est parfait. Si vous voulez implémenter d'autres fonctionnalités, méthodes ou attributs, ne vous gênez pas... mais n'oubliez pas d'y aller progressivement.

C'est parti !

Correction proposée

Voici la correction que je vous propose. Vous avez du de votre côté arriver à quelque chose, même si tout ne marche pas

parfaitement. Certaines fonctionnalités, comme le tri, l'affichage..., sont un peu complexes. Ne sautez pas trop vite à la correction cependant et essayez au moins d'obtenir un dictionnaire ordonné avec l'ajout, la consultation et la suppression d'éléments opérationnels.

ATTENTION LES YEUX...

Code : Python

```
class DictionnaireOrdonne:
    """Notre dictionnaire ordonné. L'ordre des données est maintenu
et il peut donc, contrairement aux dictionnaires usuels, être trié
ou voir l'ordre de ses données inversées.

"""
    def __init__(self, base={}, **donnees):
        """Constructeur de notre objet. Il peut ne prendre aucun
paramètre
(dans ce cas, le dictionnaire sera vide) ou construire un
dictionnaire remplis grâce :
- au dictionnaire 'base' passé en premier paramètre
- aux valeurs que l'on retrouve dans 'donnees'.

"""
        self._cles = [] # liste contenant nos clés
        self._valeurs = [] # liste contenant les valeurs
correspondantes à nos clés

        # on vérifie que 'base' est un dictionnaire exploitable
        if type(base) not in (dict, DictionnaireOrdonne):
            raise TypeError( \
                "le type attendu est un dictionnaire (usuel ou
ordonne)")

        # on récupère les données de 'base'
        for cle in base:
            self[cle] = base[cle]

        # on récupère les données de 'donnees'
        for cle in donnees:
            self[cle] = donnees[cle]

    def __repr__(self):
        """Représentation de notre objet. C'est cette chaîne qui
sera affichée
quand on entrera directement le dictionnaire dans l'interpréteur,
ou en
utilisant la fonction 'repr'.

"""
        chaine = "{"
        premier_passage = True
        for cle, valeur in self.items():
            if not premier_passage:
                chaine += ", " # on ajoute la virgule comme
séparateur
            else:
                premier_passage = False
            chaine += repr(cle) + ": " + repr(valeur)
        chaine += "}"
        return chaine

    def __str__(self):
        """Fonction appelée quand on souhaite afficher le
dictionnaire grâce
à la fonction 'print' ou le convertir en chaîne grâce au
constructeur
```

```
        'str'. On redirige sur __repr__.

        """
                return repr(self)

            def __len__(self):
                """Retourne la taille du dictionnaire"""
                return len(self._cles)

            def __contains__(self, cle):
                """Retourne True si la clé est dans la liste des clés,
        False sinon"""
                return cle in self._cles

            def __getitem__(self, cle):
                """Retourne la valeur correspondante à la clé si elle
        existe, lève
        une exception KeyError sinon.

        """
                if cle not in self._cles:
                    raise KeyError( \
                        "la clé {0} ne se trouve pas dans le
        dictionnaire".format( \
                        cle))
                else:
                    indice = self._cles.index(cle)
                    return self._valeurs[indice]

            def __setitem__(self, cle, valeur):
                """Méthode spéciale appelée quand on cherche à modifier une
        clé
        présente dans le dictionnaire. Si la clé n'est pas présente, on
        l'ajoute
        à la fin du dictionnaire.

        """
                if cle in self._cles:
                    indice = self._cles.index(cle)
                    self._valeurs[indice] = valeur
                else:
                    self._cles.append(cle)
                    self._valeurs.append(valeur)

            def __delitem__(self, cle):
                """Méthode appelée quand on souhaite supprimer une clé"""
                if cle not in self._cles:
                    raise KeyError( \
                        "la clé {0} ne se trouve pas dans le
        dictionnaire".format( \
                        cle))
                else:
                    indice = self._cles.index(cle)
                    del self._cles[indice]
                    del self._valeurs[indice]

            def __iter__(self):
                """Méthode de parcours de l'objet. On retourne l'itérateur
        des clés"""
                return iter(self._cles)

            def __add__(self, autre_objet):
                """On retourne un nouveau dictionnaire contenant les deux
        dictionnaires mis bout à bout (d'abord self puis autre_objet)

        """
                if type(autre_objet) is not type(self):
                    raise TypeError( \
                        "impossible de concaténer {0} et {1}".format( \
                        type(self), type(autre objet)))
```

```
            else:
                nouveau = DictionnaireOrdonne()

                # on commence par copier self dans le dictionnaire
                for cle, valeur in self.items():
                    nouveau[cle] = valeur

                # on copie ensuite autre_objet
                for cle, valeur in autre_objet.items():
                    nouveau[cle] = valeur
                return nouveau

    def items(self):
        """Retourne un générateur contenant les couples (cle,
valeur)"""
        for i, cle in enumerate(self._cles):
            valeur = self._valeurs[i]
            yield (cle, valeur)

    def keys(self):
        """Cette méthode retourne la liste des clés"""
        return list(self._cles)

    def values(self):
        """Cette méthode retourne la liste des valeurs"""
        return list(self._valeurs)

    def reverse(self):
        """Inversion du dictionnaire"""
        # On créée deux listes vides qui contiendront le nouvel
ordre des clés
        # et valeurs
        cles = []
        valeurs = []
        for cle, valeur in self.items():
            # On ajoute les clés et valeurs au début de la liste
            cles.insert(0, cle)
            valeurs.insert(0, valeur)
        # On met ensuite à jour nos listes
        self._cles = cles
        self._valeurs = valeurs

    def sort(self):
        """Méthode permettant de trier le dictionnaire en fonction
de ses clés"""
        # On tri les clés
        cles_triees = sorted(self._cles)
        # On créée une liste de valeurs, encore vide
        valeurs = []
        # On parcourt ensuite la liste des clés triées
        for cle in cles_triees:
            valeur = self[cle]
            valeurs.append(valeur)
        # enfin, on met à jour notre liste de clés et de valeurs
        self._cles = cles_triees
        self._valeurs = valeurs
```

Le mot de la fin

Le but de l'exercice était de présenter un énoncé à la fois simple et laissant pas mal de place aux choix de programmation. Ce que je vous propose n'est, une fois encore, pas l'unique façon de faire, ni la meilleure. L'exercice vous a surtout permi de travailler sur des notions concrètes que nous avons vu depuis le début de la partie courante, et de construire un objet conteneur qui n'est pas tout à fait dépourvu d'utilité .

N'hésitez pas à passer un peu de temps sur l'amélioration de notre objet, il n'en sera que plus joli et utile ( ) avec quelques fonctionnalités supplémentaires.

Ne vous alarmez pas trop si vous n'avez pas réussi à coder quelques aspects de notre dictionnaire. Au reste, l'essentiel est d'avoir essayé, puis d'avoir compris la correction, dans l'idéal ensuite être capable de l'améliorer.
J'espère que cette petite pause pratique vous a plu.

Le chapitre qui va suivre n'est pas obligatoire. Il présente des concepts plutôt avancés que vous n'êtes pas obligés de comprendre ou maîtriser pour coder en Python, loin de là. En fait, les chapitres cessent d'être obligatoires, ou à lire dans l'ordre, à partir de ce point.

Vous pouvez dors et déjà passer à la partie traitant de la librairie standard, si vous le désirez, et lire les chapitres qui vous intéressent particulièrement, ou basculer sur un autre tutoriel, voire une documentation  .

Et pour ceux qui souhaitent aller jusqu'au bout et découvrir la fin de cette partie sur les classes et objets, rendez-vous au prochain chapitre  !

Les décorateurs

Nous allons ici nous intéresser à un concept fascinant de Python, un concept de programmation assez avancé. Vous n'êtes pas obligé de lire ce chapitre pour continuer le tutoriel, ni même connaître cette fonctionnalité pour coder en Python. Il s'agit d'un plus que j'ai voulu détailler, mais certainement pas indispensable .

Les décorateurs sont un moyen simple de modifier les comportements "par défaut" de fonctions. C'est un exemple assez flagrant de ce qu'on appelle la **métaprogrammation**, que je vais résumer assez brièvement comme l'écriture de programmes manipulant... d'autres programmes.

Ca donne faim non ?

Qu'est-ce que c'est ?

Les décorateurs sont des fonctions de Python dont le rôle est de modifier le comportement par défaut d'autres fonctions ou classes. Pour schématiser, une fonction modifiée par un décorateur ne s'exécutera pas elle-même, mais appellera le décorateur. C'est au décorateur de décider si il veut exécuter la fonction, et dans quelle condition.

Mais quel est l'intérêt ? Si on veut juste qu'une fonction fasse quelque chose de différent, il suffit de la modifier non ? Pourquoi s'encombrer la tête avec une nouvelle fonctionnalité plus complexe ?

Il peut y avoir de nombreux cas dans lesquels les décorateurs sont un choix intéressant. Pour comprendre l'idée, je vais prendre un unique exemple :

On souhaite faire quelques tests de performance sur certaines de nos fonctions. En l'occurence, calculer combien de temps elles mettent pour s'exécuter.

Une possibilité, effectivement, est de modifier chacune des fonctions devant intégrer ce test. Mais ce n'est pas très élégant, ni très pratique, ni très sûr... bref ce n'est pas la meilleure solution.

Une possibilité est d'utiliser un décorateur. Ce décorateur se chargera d'exécuter notre fonction en calculant le temps qu'elle met et pourra afficher une alerte si le temps est trop important, par exemple.

Pour spécifier qu'une fonction doit intégrer ce test, il suffira de rajouter une simple ligne avant la définition de la fonction à tester. C'est bien plus simple, clair et adapté à la situation.

Et ce n'est qu'un exemple d'application.

Les décorateurs sont des fonctions standard de Python, mais leur construction est parfois complexe. Quand il s'agit de décorateurs prenant des arguments en paramètre, ou des décorateurs devant tenir compte des paramètres de la fonction, le code est plus complexe, moins intuitif.

Je vais faire mon possible pour que vous compreniez bien le principe. N'hésitez pas à y revenir à tête reposée, une, deux, trois fois pour que cela soit bien clair .

En théorie

Une fois n'est pas coutume, je vais vous montrer les différentes constructions possibles, en théorie, avec quelques exemples, mais je vais consacrer une sous-partie entière aux exemples d'utilisation, pour expliciter cette partie théorique mais indispensable.

Format le plus simple

Comme je l'ai dit, les décorateurs sont des fonctions "classiques" de Python, dans leur définition. Ils ont une petite subtilité, en ce qu'ils prennent en paramètre une fonction, et retournent une fonction.

On déclare qu'une fonction doit être modifiée par un (ou plusieurs) décorateurs grâce à une (ou plusieurs) lignes au-dessus de la définition de fonction, comme ceci :

Code : Python

```
@nom_du_decorateur
def ma_fonction(...)
```

Le décorateur s'exécute au moment de la définition de fonction, non pas lors de l'appel. Ceci est important. Il prend en paramètre, comme je l'ai dit, une fonction (celle qu'il modifie) et retourne une fonction (ce peut être la même).

Voyez plutôt :

Code : Python Console

```
>>> def mon_decorateur(fonction):
...     """Premier exemple de décorateur"""
...     print("Notre décorateur est appelé avec en paramètre la
fonction {0}".format(fonction))
...     return fonction
...
>>> @mon_decorateur
... def salut():
...     """Fonction modifiée par notre décorateur"""
...     print("Salut !")
...
Notre décorateur est appelé avec en paramètre la fonction <function
salut at 0x0
0BA5198>
>>>
```

Euuuuh qu'est-ce qu'on a fait là ?

- D'abord, on crée notre décorateur. Il prend en paramètre, comme je vous l'ai dit, la fonction qu'il modifie. Il se contente d'afficher cette fonction dans notre exemple, puis la retourne
- On crée ensuite notre fonction *salut*. On précise avant la définition la ligne `@mon_decorateur` qui précise à Python qu'on souhaite que cette fonction soit modifiée par notre décorateur. Notre fonction est très utile : elle affiche "Salut !" et c'est tout
- A la fin de la définition de notre fonction, on peut voir que le décorateur est appelé. Si vous regardez plus attentivement la ligne affichée, vous vous rendez compte qu'il est appelé avec, en paramètre, la fonction *salut* que nous venons de définir.

Intéressons-nous un peu plus à la structure de notre décorateur. Il prend en paramètre la fonction à modifier (celle que l'on définit sous la ligne du @), je pense que vous avez pu le constater. Mais il retourne cette fonction, également, et ça c'est un peu moins évident.

En fait, la fonction retournée remplace la fonction définie. Ici, on retourne la fonction définie, c'est donc la même. Mais on peut demander à Python d'exécuter une autre fonction à la place, pour modifier son comportement. Nous allons voir ça un peu plus bas.

Pour l'heure, souvenez-vous que les deux codes ci-dessous sont identiques :

Code : Python

```
# exemple avec décorateur
@decorateur
def fonction(...):
    ...
```

Code : Python

```
# exemple équivalent, sans décorateur
```

```
def fonction(...):
    ...

fonction = decorateur(fonction)
```

Relisez bien ces deux codes, ils font la même chose. Le second est là pour que vous compreniez ce que Python fait quand il manipule des fonctions modifiées par un (ou plusieurs) décorateur(s).

Quand vous exécutez *salut*, vous ne voyez aucun changement. Et c'est normal, puisque nous retournons la même fonction. Le seul moment où notre décorateur est appelé a été lors de la définition de notre fonction. Notre fonction *salut* n'a pas été modifiée par notre décorateur, on s'est contenté de la retourner tel qu'elle.

Modifier le comportement de notre fonction

Vous l'aurez deviné, un décorateur comme nous l'avons créé plus haut n'est pas bien utile  . Les décorateurs sont surtout utiles pour modifier le comportement d'une fonction. Je vous montre cependant pas à pas comment cela fonctionne, sinon vous risquez de vite vous perdre  .

 Comment faire pour modifier le comportement de notre fonction ?

En fait, vous avez un élément de réponse un peu plus haut. J'ai dit que notre décorateur prenait en paramètre la fonction définie, et retournait une fonction (peut-être la même, peut-être une autre). C'est cette fonction retournée qui sera directement affectée à notre fonction définie. Si vous aviez retourné une autre fonction que *salut* dans notre exemple ci-dessus, la fonction *salut* aurait redirigé vers cette fonction retournée.

 Mais alors... il faut définir encore une fonction ?

Et oui. Je vous avais prévenu (et ce n'est que le début), notre construction se complexifie au fur et à mesure : on va devoir créer une nouvelle fonction qui va être chargée de modifier le comportement de la fonction définie. Et, parce que notre décorateur sera le seul à utiliser cette fonction, on va la définir directement dans le corps de notre décorateur.

 Je suis perdu. Comment ça marche, concrètement ?

Je vais vous mettre le code, cela vaudra mieux que des tonnes d'explications. Je le commente un peu plus bas, ne vous inquiétez pas  :

Code : Python

```
def mon_decorateur(fonction):
    """Notre décorateur : il va afficher un message avant l'appel
de la
fonction définie

"""
    def fonction_modifiee():
        """Fonction que l'on va retourner. Il s'agit en fait d'une
version
un peu modifiée de notre fonction originellement définie. On se
contente d'afficher un avertissement avant d'exécuter notre
fonction
originellement définie.

"""
        print("Attention ! On appelle {0}".format(fonction))
        return fonction()
```

```
        return fonction_modifiee

    @mon_decorateur
    def salut():
        print("Salut !")
```

Voyons l'effet, avant les explications. Aucun message ne s'affiche en entrant ce code. Par contre, si vous exécutez votre fonction *salut* :

Code : Python Console

```
>>> salut()
Attention ! On appelle <function salut at 0x00BA54F8>
Salut !
>>>
```

Et si vous affichez la fonction *salut* dans l'interpréteur, vous obtenez quelque chose de surprenant :

Code : Python Console

```
>>> salut
<function fonction_modifiee at 0x00BA54B0>
>>>
```

Pour comprendre, revenons sur le code de notre décorateur :

- Comme toujours, il prend en paramètre une fonction. Cette fonction, quand on place l'appel au décorateur au-dessus de *def salut* , c'est *salut* (la fonction définie à l'origine)
- Dans le corps-même de notre décorateur, on définit une nouvelle fonction, *fonction_modifiee*. Elle ne prend aucun paramètre, elle n'en a pas besoin. Dans son corps, on affiche une ligne avertissant qu'on va exécuter la fonction *fonction* (là encore, il s'agit de *salut*). A la ligne suivante, on l'exécute effectivement et on retourne le résultat de son exécution (dans le cas de *salut*, il n'y en a pas mais d'autres fonctions pourraient retourner des informations)
- De retour dans notre décorateur, on indique devoir retourner *fonction_modifiee*.

Lors de la définition de notre fonction *salut*, on appelle notre décorateur. Python lui passe en paramètre la fonction *salut*. Cette fois, notre décorateur ne retourne pas *salut* mais *fonction_modifiee*. Et notre fonction *salut* que nous venons de définir sera donc remplacée par notre fonction *fonction_modifiee*, définie dans notre décorateur.

Vous le voyez bien d'ailleurs : quand on cherche à afficher *salut* dans l'interpréteur, on obtient *fonction_modifiee*.

Souvenez-vous bien que le code :

Code : Python

```
@mon_decorateur
def salut():
    ...
```

revient au même, pour Python, que le code :

Code : Python

```
def salut():
    ...

salut = mon_decorateur(salut)
```

Ce n'est peut-être pas plus clair. Prenez le temps de lire et de bien comprendre l'exemple. Ce n'est pas simple, la logique est bel et bien là mais il faut tester un certain temps avant d'en être sûr .

Pour résumer, notre décorateur retourne une fonction de substitution. Quand on appelle *salut*, on appelle en fait notre fonction modifiée qui appelle également *salut* après avoir affiché un petit message d'avertissement.

Autre exemple : un décorateur chargé tout simplement d'empêcher l'exécution de la fonction. Au lieu d'exécuter la fonction d'origine, on lève une exception pour avertir l'utilisateur qu'il utilise une fonctionnalité obsolète.

Code : Python

```
def obsolete(fonction_origine):
    """Décorateur levant une exception pour noter que la
fonction_origine
est obsolète.

"""
    def fonction_modifiee():
        raise RuntimeError("la fonction {0} est obsolète
!".format(fonction_origine))
    return fonction_modifiee
```

Là encore, faites quelques essais, tout deviendra limpide après quelques manipulations .

Un décorateur avec des paramètres

Toujours plus dur ! On voudrait maintenant passer des paramètres à notre décorateur. Nous allons essayer de coder un décorateur chargé d'exécuter une fonction en contrôlant le temps qu'elle met à s'exécuter. Si elle met un temps supérieur au temps passé en paramètre du décorateur, on affiche une alerte.

La ligne appelant notre décorateur, au-dessus de la définition de notre fonction, sera donc sous la forme :

Code : Python

```
@controler_temps(2.5) # 2,5 secondes maximum pour la fonction ci-
dessous
```

Jusqu'ici nos décorateurs ne prenaient aucune parenthèse après leur appel. Ces deux parenthèses sont très importantes : notre fonction de décorateur prendra en paramètre, non pas une fonction, mais les paramètres du décorateur (ici, le temps maximum autorisé pour la fonction). Elle ne retournera pas une fonction de substitution, mais un décorateur .

Encore et toujours perdu. Pourquoi est-ce si compliqué de passer des paramètres à notre décorateur ?

En fait... ce n'est pas si compliqué que ça mais c'est dûr à saisir au début. Pour mieux comprendre, essayez encore une fois de vous souvenir que ces deux codes reviennent au même :

Code : Python

```
@decorateur
```

```
def fonction(...):
    ...
```

Code : Python

```
def fonction(...):
    ...

fonction = decorateur(fonction)
```

C'est la dernière ligne du second exemple que vous devez retenir et essayer de comprendre : `fonction = decorateur(fonction)`.

On remplace la fonction que nous avons définie au-dessus par la fonction que retourne notre décorateur.

C'est le mécanisme qui se cache derrière notre *@decorateur*.

Maintenant, si notre décorateur attend des paramètres. On se retrouve avec une ligne comme celle-ci :

Code : Python

```
@decorateur(parametre)
def fonction(...):
    ...
```

Et si vous avez compris l'exemple ci-dessus, ce code revient au même que :

Code : Python

```
def fonction(...):
    ...

fonction = decorateur(parametre)(fonction)
```

Je vous avais prévenu, ce n'est pas très intuitif. Mais relisez bien ces exemples, le déclic devrait se faire tôt ou tard  .

Comme vous le voyez (du moins, j'espère ), on doit définir, comme décorateur, une fonction qui prend en paramètre les paramètres du décorateur (ici, le temps attendu) et qui retourne un décorateur. Autrement dit on se retrouve encore une fois avec un niveau supplémentaire dans notre fonction.

Je vous donne le code sans trop insister. Si vous arrivez à comprendre la logique qui se trouve derrière, c'est tant mieux, sinon n'hésitez pas à y revenir plus tard  :

Code : Python

```
"""Pour gérer le temps, on importe le module time
On va utiliser surtout la fonction time() de ce module qui retourne
le nombre
de secondes depuis le premier janvier 1970 (habituellement).
On va s'en servir pour calculer le temps mis par notre fonction
pour
s'exécuter.

"""
```

```
import time

def controler_temps(nb_secs):
    """Contrôle le temps mis par une fonction pour s'exécuter.
Si le temps d'exécution est supérieur à nb_secs, on affiche une
alerte.

"""
    def decorateur(fonction_a_executer):
        """Notre décorateur. C'est lui qui est appelé directement
LORS
DE LA DEFINITION de notre fonction (fonction_a_executer)

"""
        def fonction_modifiee():
            """Fonction retournée par notre décorateur. Elle se
charge
de calculer le temps mis par la fonction à exécuter.

"""
            tps_avant = time.time() # avant d'exécuter la fonction
            valeur_retournee = fonction_a_executer() # on exécute la
fonction
            tps_apres = time.time()
            tps_execution = tps_apres - tps_avant
            if tps_execution >= nb_secs:
                print("La fonction {0} a mis {1} pour
s'exécuter".format( \
                            fonction_a_executer, tps_execution))
            return valeur_retournee
        return fonction_modifiee
    return decorateur
```

Ouf ! Trois niveaux dans notre fonction ! D'abord *controler_temps* qui définit dans son corps notre décorateur *decorateur* qui définit lui-même dans son corps notre fonction modifiée *fonction_modifiee*.

J'espère que vous n'êtes pas trop embrouillé. Je le répète, il s'agit d'une fonctionnalité très puissante mais qui n'est pas très intuitive, quand on n'y est pas habitué. Jetez un oeil du côté des exemples au-dessus si vous êtes un peu perdu.

Nous pouvons maintenant utiliser notre décorateur. J'ai fait une petite fonction pour tester qu'un message s'affiche bien si notre fonction met du temps à s'exécuter. Voyez plutôt :

Code : Python Console

```
>>> @controler_temps(4)
... def attendre():
...     input("Appuyez sur Entrée...")
...
>>> attendre() # je vais appuyer sur Entrée presque tout de suite
Appuyez sur Entrée...
>>> attendre() # Cette fois, j'attends plus longtemps
Appuyez sur Entrée...
La fonction <function attendre at 0x00BA5810> a mis 4.14100003242
pour s'exécute
r
>>>
```

Ca marche ! Et, même si vous devez passer un peu de temps sur votre décorateur vu ses différents niveaux, vous êtes obligé de reconnaître qu'il s'utilise des plus simplement .

Il est quand même plus intuitif d'écrire :

Code : Python

```
@controler_temps(4)
def attendre(...)
    ...
```

plutôt que :

Code : Python

```
def attendre(...):
    ...

attendre = controler_temps(4)(attendre)
```

Ces deux codes reviennent au même, mais le premier est quand même plus intuitif. Je vous mets le second pour vous aider à comprendre ce que nous avons fait, et pourquoi.

Tenir compte des paramètres de notre fonction

Jusqu'ici, nous n'avons travaillé qu'avec des fonctions ne prenant aucun paramètre. C'est pourquoi notre fonction *fonction_modifiee* n'en prenait pas non plus.

Oui mais... tenir compte des paramètres, ce peut être utile. Sans quoi on ne pourrait construire des décorateurs ne s'appliquant qu'à des fonctions sans paramètre.

Il faut, pour tenir compte des paramètres de la fonction, modifier ceux de notre fonction *fonction_modifiee*. Là encore, je vous invite à regarder les exemples ci-dessus, explicitant ce que Python fait implicitement quand on définit un décorateur avant une fonction. Vous pourrez vous rendre compte que *fonction_modifiee* remplace notre fonction et que donc, elle doit prendre des paramètres, si notre fonction définie prend également des paramètres.

C'est dans ce cas en particulier que nous allons pouvoir réutiliser la notation spéciale pour nos fonctions attendant un nombre d'arguments variable. En effet, le décorateur que nous avons créé un peu plus haut devrait pouvoir s'appliquer à des fonctions ne prenant aucun paramètre, ou en prenant un, ou plusieurs... au fond notre décorateur ne doit pas savoir combien de paramètres sont fournis à notre fonction, et ne doit pas s'en soucier.

Là encore, je vous place le code modifié de notre fonction modifiée (souvenez-vous qu'elle est définie dans notre *decorateur*, lui-même défini dans *controler_temps* (je ne vous remet le code que de *fonction_modifiee*).

Code : Python

```
...
        def fonction_modifiee(*parametres_non_nommes,
**parametres_nommes):
            """Fonction retournée par notre décorateur. Elle se
charge
de calculer le temps mis par la fonction à exécuter.

"""
            tps_avant = time.time() # avant d'exécuter la fonction
            ret = fonction_a_executer(*parametres_non_nommes,
**parametres_nommes)
            tps_apres = time.time()
            tps_execution = tps_apres - tps_avant
            if tps_execution >= nb_secs:
                print("La fonction {0} a mis {1} pour
s'exécuter".format( \
                            fonction_a_executer, tps_execution))
            return ret
```

A présent, vous pouvez appliquer ce décorateur à des fonctions ne prenant aucun paramètre, ou en prenant un certain nombre, des nommés ou des non nommés. Pratique non ?

Des décorateurs s'appliquant aux définitions de classes

Vous pouvez également appliquer des décorateurs à la définition des classes. Nous verrons un exemple d'application dans la sous-partie suivante. Au lieu de recevoir en paramètre la fonction, vous allez recevoir la classe.

Code : Python Console

```
>>> def decorateur(classe):
...     print("Définition de la classe {0}".format(classe))
...     return classe
...
>>> @decorateur
... class Test:
...     pass
...
Définition de la classe <class '__main__.Test'>
>>>
```

Voilà. Vous verrez dans la sous-partie suivante quel peut être l'intérêt de manipuler nos définitions de classes grâce à des décorateurs. Il existe d'autres exemples que celui que je vais vous montrer, bien entendu .

Chaîner nos décorateurs

Vous pouvez modifier une fonction ou une définition de classe par plusieurs décorateurs, sous la forme :

Code : Python

```
@decorateur1
@decorateur2
def fonction():
```

Ce n'est pas plus compliqué que ce que vous venez de faire. Je vous le montre pour que ne subsiste aucun doute dans votre esprit, vous pouvez tester à loisir cette possibilité, par vous-même .

Je vais à présent vous montrer quelques idées d'application des décorateurs, inspirées en grande partie de la PEP 318 -- Decorators for Functions and Methods.

Exemples d'application

Nous allons voir deux exemples d'application des décorateurs dans cette sous-partie. Vous en avez également vu quelques uns, plus ou moins utilisables, dans la sous-partie précédente .

Des classes singleton

Certains reconnaîtront sûrement cette appellation. Pour les autres, sachez qu'une classe dite **singleton** est une classe qui ne peut être instanciée qu'une fois.

Autrement dit, on ne peut créer qu'un seul objet de cette classe.

Cela peut-être utile parfois, quand vous voulez être absolument certain qu'une classe ne produira qu'un seul objet, qu'il est inutile (voire dangereux) d'avoir plusieurs objets de cette classe. La première fois que vous appelez le constructeur de ce type de classe, on obtient le premier et l'unique objet nouvellement instancié. La seconde fois, comme toutes les autres fois par la suite, qu'on appelle ce constructeur, on obtient le même objet (le premier créé).

Ceci est très facile à modéliser grâce à des décorateurs.

Code de l'exemple

Code : Python

```
def singleton(classe_definie):
    instances = {} # dictionnaire de nos instances singleton
    def get_instance():
        if classe_definie not in instances:
            # on crée notre premier objet de classe_definiee
            instances[classe_definie] = classe_definie()
        return instances[classe_definie]
    return get_instance
```

Explications

D'abord, pour utiliser notre décorateur. C'est très simple, il suffit de mettre l'appel à notre décorateur avant la définition des classes que nous souhaitons singleton :

Code : Python Console

```
>>> @singleton
... class Test:
...     pass
...
>>> a = Test()
>>> b = Test()
>>> a is b
True
>>>
```

Quand on crée notre premier objet (celui se trouvant dans *a*), notre constructeur est bien appelé. Quand on souhaite créer un second objet, c'est celui contenu dans *a* qui est retourné. Ainsi, *a* et *b* pointent vers le même objet  .

Intéressons-nous maintenant à notre décorateur. Il définit dans son corps un dictionnaire. Ce dictionnaire contient en clé la classe singleton, et en valeur l'objet créé correspondant. Il retourne notre fonction interne *get_instance* qui va remplacer notre classe. Ainsi, quand on voudra créer un nouvel objet, ce sera *get_instance* qui sera appelée. Cette fonction vérifie si notre classe se trouve dans le dictionnaire. Si ce n'est pas le cas, on crée notre premier objet correspondant et on l'ajoute dans le dictionnaire. Dans tous les cas, on retourne l'objet correspondant dans le dictionnaire (il vient soit d'être créé, soit c'est notre objet créé au premier appel du constructeur).

Grâce à ce système, on peut avoir plusieurs classes déclarées comme des singleton, et on est sûr que pour chacune de ces classes un seul objet sera créé.

Contrôler les types passés à notre fonction

Vous l'avez déjà observé dans Python : aucun contrôle n'est fait sur le type des données passées en paramètre de nos fonctions. Certaines, comme *print*, acceptent n'importe quel type. D'autres lèvent des exceptions quand un paramètre d'un type incorrect leur est fourni.

Il pourrait être utile de coder un décorateur qui vérifie les types passés en paramètre de notre fonction et qui lève une exception si les types attendus ne correspondent pas à ceux reçus lors de l'appel à la fonction.

Voici notre définition de fonction, pour vous donner une idée :

Code : Python

```
@controler types(int, int)
```

```
def intervalle(base_inf, base_sup):
```

Notre décorateur *controler_types* doit s'assurer qu'à chaque fois qu'on appelle la fonction *intervalle*, ce sont des entiers qui sont passés en paramètre en tant que *base_inf* et *base_sup*.

Ce décorateur est plus complexe, bien que j'ai simplifié au maximum l'exemple de la PEP 318.

Encore une fois, si il est un peu long à écrire, il est d'une simplicité enfantine à utiliser  .

Code de l'exemple

Code : Python

```
def controler_types(*a_args, **a_kwargs):
    """On attend en paramètre du décorateur les types attendus. On
accepte
une liste de paramètres indéterminés, étant donné que notre
fonction
définie pourra être appelée avec un nombre variable de paramètres,
et que
chacun doit être contrôlé.

"""
    def decorateur(fonction_a_executer):
        """Notre décorateur. Il doit retourner fonction_modifiee"""
        def fonction_modifiee(*args, **kwargs):
            """Notre fonction modifiée. Elle se charge de contrôler
les types qu'on lui passe en paramètre.

"""
            # la liste des paramètres attendus (a_args) doit être
de même
            # longueur que celle reçue (args)
            if len(a_args) != len(args):
                raise TypeError("le nombre d'arguments attendu n'est
pas égal " \
                                "au nombre reçu")
            # on parcourt la liste des arguments reçus et non nommé
s
            for i, arg in enumerate(args):
                if a_args[i] is not type(args[i]):
                    raise TypeError("l'argument {0} n'est pas du
type " \
                                    "{1}".format(i, a_args[i]))

            # on parcourt à présent la liste des paramètres reçus
et nommés
            for cle in kwargs:
                if cle not in a_kwargs:
                    raise TypeError("l'argument {0} n'a aucun type "
\
                                    "précisé".format(repr(cle)))
                if a_kwargs[cle] is not type(kwargs[cle]):
                    raise TypeError("l'argument {0} n'est pas de
type" \
                                    "{1}".format(repr(cle), a_kwargs[cle]))
            return fonction_a_executer(*args, **kwargs)
        return fonction_modifiee
    return decorateur
```

Explications

C'est un décorateur assez complexe (et pourtant croyez-moi je l'ai simplifié autant que possible). Nous allons d'abord voir comment l'utiliser :

Code : Python Console

```
>>> @controler_types(int, int)
... def intervalle(base_inf, base_sup):
...     print("Intervalle de {0} à {1}".format(base_inf, base_sup))
...
>>> intervalle(1, 8)
Intervalle de 1 à 8
>>> intervalle(5, "oups!")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 24, in fonction_modifiee
TypeError: l'argument 1 n'est pas du type <class 'int'>
>>>
```

Là encore, l'utilisation est des plus simple. Intéressons-nous au décorateur proprement dit c'est déjà un peu plus complexe) :

Notre décorateur doit prendre des paramètres (une liste de paramètres indéterminés d'ailleurs, car notre fonction doit prendre une liste indéterminée de paramètres également et qu'on doit contrôler chaque paramètre). On définit donc un paramètre *a_args* qui contient la liste des types des paramètres non nommés attendus, et un second paramètre *a_kwargs* qui contient le dictionnaire des types des paramètres nommés attendus.

Vous suivez toujours ?

Vous devriez comprendre la construction d'ensemble, nous l'avons vu un peu plus haut. Elle comprend trois niveaux, puisque nous devons influer sur le comportement de la fonction et que notre décorateur prend des paramètres. Notre code de contrôle se trouve, comme il se doit, dans notre fonction *fonction_modifiee* (qui va prendre la place de notre *fonction_a_executer*).

On commence par vérifier que la liste des paramètres non nommés attendus est bien égale à la liste des paramètres non nommés reçus, en taille. On vérifie ensuite individuellement chaque paramètre reçu, en contrôlant son type. Si le type reçu est égal au type attendu, tout va bien. Sinon, on lève une exception. On répète l'opération sur les paramètres nommés (avec une petite différence, puisqu'il s'agit de paramètres nommés ils sont contenus dans un dictionnaire, pas une liste).

Si tout va bien (aucune exception n'a été levée), on exécute notre fonction en retournant son résultat.

Voilà nos exemples d'application. Il y en a bien d'autres, vous pouvez en retrouver plusieurs sur la PEP 318 consacrée aux décorateurs, ainsi que des informations supplémentaires, n'hésitez pas à y faire un petit tour .
Vous savez à présent comment utiliser les décorateurs. Ce n'est pas un chapitre très simple, n'hésitez pas à y revenir si le sujet vous intéresse et que vous n'avez pas tout saisi. Je pense qu'il est nécessaire de lire ce chapitre a tête reposée et ne pas hésiter à faire pas mal d'essais de son côté avant de bien maîtriser le sujet.

Ce chapitre conclut cette troisième partie consacrée aux objets. Non pas que vous ayez tout vu... mais je m'arrête ici, pour ma part. Vous trouverez d'autres tutoriels pour aller plus loin, et, surtout, la documentation de Python, qui sera bien plus complète que ce que je pourrais jamais écrire. Bonne chance, et rendez-vous dans la prochaine partie, à lire dans le désordre si vous le désirez , présentant quelques modules intéressants de la librairie standard.

Encore un tour d'horizon bouclé . Vous avez à présent un niveau suffisant en Python pour vous lancer dans de nombreux projets. La prochaine partie est là pour vous y aider, mais vous devrez tôt ou tard vous pencher sur des documentations si vous voulez progresser.

Tout ce que vous avez appris jusqu'à présent vous sera utile d'une façon générale. Pour vous spécialiser, je vous propose un petit tour du côté de la librairie standard. Vous pourrez également consulter d'autres tutoriels ou documentations sur des librairies tierces. Mais dors et déjà, vos connaissances sont suffisantes pour programmer en Python à un bon niveau et avec des bons réflexes .

Sachez enfin que je n'ai pas pu traiter tout l'orienté objet côté développeur dans cette partie. Il vous reste des choses à apprendre, même si elles sont moins utilisées, ou moins utiles dans un contexte général. Les connaissances que j'ai essayé de vous transmettre vous suffiront largement pour aller plus loin, si vous le désirez.

Partie 4 : Les merveilles de la librairie standard

Cette partie consiste en une présentation de quelques modules intéressants de la librairie standard que j'ai voulu souligner. Vous n'y apprendrez pas à installer et contrôler des librairies tierces, et cette partie ne se veut pas une présentation exaustive de tous les modules de la librairie standard : ce serai trop long, et la documentation officielle est faite pour ça .

En attendant je vous propose d'aborder quelques thèmes intéressants, organisés en chapitre que vous pouvez lire dans le désordre. En début de chaque chapitre je préciserai les points que vous devez maîtriser pour vous lancer dans l'approche du sujet. Je ne vous conseille pas de lire ces chapitres avant d'avoir lu la première partie de ce tutoriel.

De plus, je vous donnerai souvent des moyens d'aller plus loin si ce sujet vous intéresse, en vous renvoyant la plupart du temps à la documentation officielle, en anglais mais, encore une fois, bien plus complète que ce que vous pouvez espérer trouver ailleurs.

Les expressions régulières

Dans ce chapitre, je vais m'attarder sur les **expressions régulières** et sur le module *re* qui permet de les manipuler. Si vous ne savez pas ce que sont les expressions régulières, je l'explique un peu plus bas.

Il existe, naturellement, bien d'autres modules permettant de manipuler du texte. C'est toutefois sur celui-ci que je vais m'attarder aujourd'hui, mais je vais vous donner les moyens d'aller plus loin si vous le désirez .

Que sont les expressions régulières ?

Les **expressions régulières** sont un puissant moyen de rechercher et isoler des expressions d'une chaîne de caractère.

Pour simplifier, imaginez que vous faites un programme qui demande un certain nombre d'informations à l'utilisateur pour, par exemple, les stocker dans un fichier. Pour demander son nom, son prénom et quelques autres informations, ce n'est pas bien difficile : on va utiliser la fonction *input* et récupérer le résultat. Jusqu'ici rien de nouveau.

Mais si on demande à l'utilisateur d'entrer un numéro de téléphone ? Qu'est-ce qui l'empêche d'entrer n'importe quoi ? Si on lui demande d'entrer une adresse e-mail et qu'il en entre une invalide, comme par exemple *"je_te_donnerai_pas_mon_adresse_mail"* et qu'on souhaite envoyer automatiquement un mail à cette personne ? Que va-t-il se passer ?

Si ce cas n'est pas géré, vous risquez d'avoir un problème. Les expressions régulières sont un moyen de rechercher, isoler ou remplacer des expressions dans une chaîne. Ici, elles nous permettraient de vérifier que le numéro de téléphone entré compte bien dix chiffres, qu'il commence par un 0 et qu'il compte éventuellement des séparateurs tous les deux chiffres. Si ce n'est pas le cas, on demande à l'utilisateur de l'entrer de nouveau.

Quelques syntaxes pour les expressions régulières

Si vous connaissez déjà les expressions régulières et leur syntaxe, vous pouvez passer au point sur le module *re*. Sinon, sachez que je me contenterais d'une approche des expressions régulières. C'est un sujet trop vaste pour que j'y passe si peu de temps. Ne paniquez pas toutefois, je vais vous donner quelques exemples concrets et vous pourrez toujours trouvez d'autres explications de part le Web .

Concrètement, ça se présente comment ?

Le module *re* que nous allons découvrir un peu plus bas nous permet de rechercher de façon très précise dans des chaînes de caractères et de remplacer des éléments de nos chaînes en fonction de critères particuliers. Ces critères, ce sont nos **expressions régulières**. Elles vont être pour nous sous la forme de chaînes de caractères. Les expressions régulières deviennent assez rapidement dures à lire, mais ne vous en faites pas : nous allons y aller petit à petit .

Des caractères ordinaires

Quand on forme une expression régulière, on peut utiliser des caractères spéciaux et d'autres qui ne le sont pas. Par exemple, si nous recherchons le mot *chat* dans notre chaîne, nous pouvons entrer comme expression régulière la chaîne *"chat"*. Jusque-là, rien de très compliqué .

Mais vous vous doutez bien que les expressions régulières ne se limitent pas à ce type de recherche extrêmement simple, sans quoi les méthodes *find* et *replace* de la classe *str* auraient suffis.

Rechercher au début ou à la fin de la chaîne

Vous pouvez rechercher au début de la chaîne en plaçant au début de votre regex (abréviation de **Regular Expression**) le signe d'accent circonflexe ^. Si par exemple vous voulez rechercher la syllabe *cha* en début de votre chaîne, vous entrerez l'expression *^cha*. Cette expression sera trouvée dans la chaîne *'chaton'*, mais pas dans la chaîne *'achat'*.

Pour matérialiser la fin de la chaîne, vous utiliserez le signe *$*. L'expression *q$* sera trouvée uniquement si votre chaîne se termine par la lettre q minuscule.

Contrôler le nombre d'occurences

Les caractères spéciaux que nous allons découvrir permettent de contrôler le nombre de fois où notre expression apparaît dans notre chaîne.

Regardez l'exemple ci-dessous :

Code : Autre

```
chat*
```

Nous avons rajouté un signe astérisque (*) après notre *t* de *chat*. Cela signifie que notre lettre *t* pourra se retrouver 0, 1, 2, ... fois dans notre chaîne. Autrement dit, notre expression *chat** sera trouvée dans les chaînes suivantes : *'chat'*, *'chaton'*, *'chateau'*, *'herbe à chat'*, *'chapeau'*, *'chatterton'*, *'chatttttttt'*...

Regardez un à un les exemples ci-dessus pour vérifier que vous les comprenez bien. On trouvera dans chacune des chaînes ci-dessus l'expression régulière *chat**. Traduit en français cette expression signifie : une lettre *c* suivie d'une lettre *h* suivie d'une lettre *a* suivie, éventuellement, d'une lettre *t* qu'on peut trouver zéro, une ou plusieurs fois. Peu importe que ces lettres soient trouvées au début, à la fin ou au milieu de la chaîne.

Un autre exemple ? Considérez l'expression régulière ci-dessous et essayez de la comprendre :

Code : Autre

```
bat*e
```

Cette expression est trouvée dans les chaînes suivantes : *'bateau'*, *'batteur'* et *'joan baez'*  .

Dans nos exemples, le signe * n'agit que sur la lettre qui le précède, pas sur les autres avant ni sur celles après.

Il existe d'autres signes permettant de contrôler l'occurence :

Signe	Explication	Exemple d'expression	Exemples de chaînes contenant l'expression
*	Vous venez de le voir. Il signifie 0, 1 ou plus	abc*	*'ab'*, *'abc'*, *'abcc'*, *'abccccc'*
+	1 ou +	abc+	*'abc'*, *'abcc'*, *'abccc'*
?	0 ou 1	abc?	*'ab'*, *'abc'*

Vous pouvez également contrôler précisément le nombre d'occurences grâce aux accolades :

- *E{4}* : signifie 4 fois la lettre E majuscule
- *E{2,4}* : signifie de 2 à 4 fois la lettre E majuscule
- *E{,5}* : signifie de 0 à 5 fois la lettre E majuscule
- *E{8,}* : signifie 8 fois minimum la lettre E majuscule.

Les classes de caractères

Vous pouvez préciser entre crochets plusieurs caractères ou classes de caractères. Par exemple, si vous entrez *[abcd]* cela signifie : soit la lettre *a*, *b*, *c* ou *d*.

Pour exprimer des classes, vous pouvez utiliser le tiret - entre deux lettres. Par exemple, l'expression *[A-Z]* signifie "une lettre majuscule". Vous pouvez préciser plusieurs classes ou possibilités dans votre expression.

Par exemple l'expression *[A-Za-z0-9]* signifie "une lettre, majuscule ou minuscule, ou un chiffre".

Vous pouvez aussi contrôler l'occurence des classes comme nous l'avons vu juste au-dessus. Si vous voulez par exemple rechercher 5 lettres majuscules se suivant dans une chaîne, votre expression sera *[A-Z]{5}*.

Les groupes

Je vous donne beaucoup de choses à retenir et vous n'avez pas encore l'occasion de pratiquer. C'est le dernier point sur lequel je vais m'attarder et il va être rapide : comme je l'ai dit plus haut si vous voulez par exemple contrôler l'occurence d'un caractère, vous ajoutez derrière un signe particulier (un astérisque, un point d'interrogation, des accolades...). Mais si vous voulez appliquer ce contrôle d'occurence sur plusieurs caractères, vous allez mettre ces caractères entre parenthèses.

Code : Autre

```
(cha){2,5}
```

Cette expression sera vraie pour les chaînes contenant *'cha'* répétée entre deux et cinq fois. Les *'cha'* doivent se suivre naturellement  .

Les groupes sont également utiles pour remplacer des portions de notre chaîne mais nous y reviendront plus tard, quand sera venue l'heure de la pratique... hum ? C'est l'heure ? Ah bah c'est parti, alors  !

Le module re

Le module *re* a été spécialement conçu pour travailler avec les expressions régulières (**Regular Expressions**). Il définit plusieurs fonctions utiles, que nous allons découvrir, ainsi que des objets propres, pour modéliser des expressions.

Chercher dans une chaîne

Nous allons pour ce faire utiliser la fonction *search* du module *re*. Bien entendu , pour pouvoir l'utiliser il faut l'importer.

Code : Python Console

```
>>> import re
>>>
```

Cette fonction attend deux paramètres obligatoires : l'expression régulière, sous la forme d'une chaîne, et la chaîne dans laquelle on recherche cette expression. Si l'expression est trouvée, la fonction retourne un objet symbolisant l'expression recherchée. Sinon, elle retourne *None*.

Note importante :

Certains caractères spéciaux dans nos expressions régulières sont modélisés par l'anti-slash \. Vous n'êtes pas sans savoir que, de son côté, Python représente d'autres caractères avec ce symbole. Si vous entrez dans une chaîne *\n*, cela voudra dire pour Python un saut de ligne.

Pour symboliser les caractères spéciaux dans les expressions régulières, il est nécessaire d'échapper l'anti-slash en mettant un

autre anti-slash devant. Cela veut dire que pour écrire le caractère spécial *\w*, vous allez devoir écrire *\\w*.

C'est assez peu pratique et parfois c'est gênant pour la lisibilité. C'est pourquoi je vous conseille d'utiliser un format de chaîne que nous n'avons pas vu jusqu'à présent : en mettant un *r* avant le délimiteur ouvrant notre chaîne, tous les anti-slashes la contenant sont échappés.

Code : Python Console

```
>>> r'\n'
'\\n'
>>>
```

Si vous avez du mal à voir l'intérêt, je vous conseille simplement de vous rappeler de mettre un *r* avant d'écrire des chaînes contenant des expressions, comme vous allez le voir dans les exemples que je vais vous donner.

Mais revenons à notre fonction *search*. Nous allons mettre en pratique ce que nous avons vu au-dessus :

Code : Python Console

```
>>> re.search(r"abc", "abcdef")
<_sre.SRE_Match object at 0x00AC1640>
>>> re.search(r"abc", "abacadaeaf")
>>> re.search(r"abc*", "ab")
<_sre.SRE_Match object at 0x00AC1800>
>>> re.search(r"abc*", "abccc")
<_sre.SRE_Match object at 0x00AC1640>
>>> re.search(r"chat*", "chateau")
<_sre.SRE_Match object at 0x00AC1800>
>>>
```

Comme vous le voyez, si l'expression est trouvée dans la chaîne, un objet de la classe *_sre.SRE_Match* est retourné. Si l'expression n'est pas trouvée, la fonction retourne *None*.

Ce qui fait qu'il est extrêmement facile de savoir si une expression est contenue dans une chaîne :

Code : Python

```
if re.match(expression, chaine) is not None:
    # si l'expression est dans la chaîne
    # ou alors, plus intuitivement
if re.match(expression, chaine):
```

N'hésitez pas à tester des syntaxes plus complexes et plus utiles. Tiens, par exemple, comment obliger l'utilisateur à entrer un numéro de téléphone ?

Avec le bref descriptif que je vous ai donné dans ce chapitre, vous pouvez, théoriquement, y arriver. Mais c'est quand même une regex assez complexe, je vous la donne, prenez le temps de la décortiquer si vous le souhaitez :

Notre regex doit vérifier qu'une chaîne est un numéro de téléphone. L'utilisateur peut entrer un numéro de différentes façons :

- 0X XX XX XX XX
- 0X-XX-XX-XX-XX
- 0X.XX.XX.XX.XX
- 0XXXXXXXXX

Autrement dit :

- Le premier chiffre doit être un 0
- Le second chiffre, ainsi que tous ceux qui suivent (9 en tout, sans compter le 0 d'origine) doivent être entre 0 et 9
- Tous les deux chiffres, on peut avoir un délimiteur optionnel (un tiret, un point ou un espace).

Voici la regex que je vous propose :

Code : Autre

```
^0[0-9]([ .-]?[0-9]{2}){4}$
```

ARGH ! C'est illisible ton truc !

Je reconnais que c'est assez peu clair. Je la décompose petit à petit :

- D'abord, le signe d'accent circonflexe ^ qui veut dire qu'on cherche l'expression au début de la chaîne. Vous pouvez voir le symbole *$* qui veut dire que l'expression doit être à la fin de la chaîne. Si l'expression doit être au début et à la fin de la chaîne, cela signifie que la chaîne dans laquelle on recherche ne doit contenir que l'expression
- Nous avons ensuite le *0* qui veut simplement dire que le premier caractère de notre chaîne doit être un *0* 
- Nous avons ensuite une classe de caractère *[0-9]*. Cela signifie qu'après le *0*, on doit trouver un nombre entre 0 et 9 (peut-être 0, peut-être 1, peut-être 2...)
- Ensuite ça se complique. Vous avez une parenthèse qui matérialise le début d'un groupe. Dans ce groupe, nous trouvons, dans l'ordre :
  - D'abord une classe *[.-]* qui veut dire "soit un espace, soit un point, soit un tiret". Juste après cette classe vous avez un signe *?* qui dit que cette classe est optionnelle
  - Après la définition de notre délimiteur, nous trouvons une classe *[0-9]* qui signifie encore une fois "un chiffre entre 0 et 9". Après cette classe, entre accolades, vous pouvez voir le nombre de chiffres attendus (2).

  Ce groupe, contenant un séparateur optionnel et deux chiffres, doit se retrouver quatre fois dans notre expression (après la parenthèse fermante, vous trouvez entre accolades le contrôle du nombre d'occurences).

Si vous regardez bien nos numéros de téléphone, vous vous rendez compte que notre regex s'applique aux différents cas présentés. La définition de notre numéro de téléphone n'est pas vraie pour tous les numéros. Cette regex est un exemple et même une base pour vous permettre de saisir le concept.

Si vous voulez que l'utilisateur entre un numéro de téléphone, voici le code auquel vous pourriez arriver :

Code : Python

```
import re
chaine = ""
expression = r"^0[0-9]([ .-]?[0-9]{2}){4}$"
while re.search(expression, chaine) is None:
    chaine = input("Entrez un numéro de téléphone (valide) :")
```

Remplacer une expression

Le remplacement est un peu plus complexe. Je ne vais pas vous montrer d'exemples réellement utiles, car ils s'appuient en général

sur des expressions assez difficiles à comprendre.

Pour remplacer des parties de nos expressions par d'autres, nous allons utiliser la fonction *sub* du module *re*.

Elle prend trois paramètres :

- L'expression à rechercher
- Par quoi remplacer cette expression
- La chaîne d'origine.

Elle retourne la chaîne modifiée.

Des groupes numérotés

Pour remplacer une partie de l'expression, on doit d'abord utiliser des groupes. Si vous vous rappelez, les groupes sont indiqués entre parenthèses.

Code : Autre

```
(a)b(cd)
```

Dans cet exemple, *(a)* est le premier groupe et *(cd)* est le second.

L'ordre des groupes est important dans cet exemple. Dans notre expression de remplacement, on peut appeler nos groupes grâce à \<*numéro du groupe*>. Pour une fois, on compte à partir de 1 .

Ce n'est pas très clair ? Regardez cet exemple simple :

Code : Python Console

```
>>> re.sub(r"(ab)", r" \1 ", "abcdef")
' ab cdef'
>>>
```

On se contente dans cet exemple de remplacer *'ab'* par *' ab '*.

Je vous l'accorde, on serait parvenu au même résultat en utilisant la méthode *replace* de notre chaîne. Mais les expressions régulières sont bien plus précises que ça , vous commencez à vous en rendre compte je pense.

Je vous laisse le soin de creuser la question, je ne préfère pas vous présenter tout de suite des expressions trop complexes .

Donner des noms à nos groupes

On peut également donner des noms à nos groupes. Cela peut être plus clair que de compter sur des numéros. Il faut pour cela faire suivre la parenthèse ouvrant le groupe d'un point d'interrogation, d'un P majuscule et du nom du groupe entre chevrons <>.

Code : Autre

```
(?P<id>[0-9]{2})
```

Dans l'expression de remplacement, on utilisera l'expression \g<*nom du groupe*> pour symboliser le groupe. Prenons un exemple :

Code : Python Console

```
>>> texte = """
... nom='Task1', id=8
... nom='Task2', id=31
... nom='Task3', id=127
... ...
... """
>>> print(re.sub(r"id=(?P<id>[0-9]+)", r"id[\g<id>]", texte))
nom='Task1', id[8]
nom='Task2', id[31]
nom='Task3', id[127]
...
>>>
```

Des expressions compilées

Si vous utilisez plusieurs fois dans votre programme les mêmes expressions régulières, il peut être utile de les compiler. Le module *re* propose en effet de conserver votre expression régulière sous la forme d'un objet que vous pouvez stocker dans votre programme. Si vous devez chercher cette expression dans une chaîne, vous passez par des méthodes de l'expression. Cela vous fait gagner en performance si vous faites beaucoup appel à cette expression.

Par exemple, j'ai une expression qui est appelée quand l'utilisateur entre son mot de passe. Je veux vérifier que son mot de passe fait bien six caractères au minimum et ne contient que des lettres majuscules, minuscules et des chiffres. Voici l'expression à laquelle j'arrive :

Code : Autre

```
^[A-Za-z0-9]{6,}$
```

A chaque fois qu'un utilisateur entre un mot de passe, le programme va appeler *re.search* pour vérifier que le mot de passe respecte bien les critères de l'expression. Il serait plus judicieux de conserver l'expression en mémoire.

On utilise pour ce faire la méthode *compile* du module *re*. On stocke la valeur retournée (une expression régulière compilée) dans une variable, c'est un objet standard pour le reste.

Code : Python

```
chn_mdp = r"^[A-Za-z0-9]{6,}$"
exp_mdp = re.compile(chn_mdp)
```

Ensuite, vous pouvez utiliser cette expression compilée directement. Elle possède plusieurs méthodes utiles, dont *search* et *sub* que nous avons vu plus haut. A la différence des fonctions du module *re* portant le même nom, elles ne prennent pas en premier paramètre l'expression (celle-ci se trouve dans l'objet directement).

Voyez plutôt :

Code : Python

```
chn_mdp = r"^[A-Za-z0-9]{6,}$"
exp_mdp = re.compile(chn_mdp)
mot_de_passe = ""
while exp_mdp.search(mot_de_passe) is None:
    mot_de_passe = input("Entrez votre mot de passe : ")
```

Je n'ai pas pu faire le tour de ce module. Il reste bien des choses à découvrir, d'abord dans la syntaxes des expressions régulières, ensuite dans le module *re*.

Si vous souhaitez en apprendre d'avantage, je vous renvoie à la documentation officielle de Python et plus précisément à la partie traitant du module re. Bonne lecture ! 

Les temps

Exprimer un temps en informatique, cela soulève quelques questions. Il y a pas mal d'utilités à avoir accès à un temps dans un programme, pour connaître la date et l'heure actuelle et faire remonter une erreur, pour calculer depuis combien de temps le programme a été lancé, gérer des alarmes programmées, faire des tests de performance... et j'en passe .

Il existe plusieurs façons de représenter des temps, que nous allons découvrir maintenant.

Pour ce chapitre, je vous conseille d'une façon générale de savoir ce qu'est un objet et comment en créer un .

Le module time

Le module *time* est sans doute le premier à être utilisé quand on souhaite manipuler du temps de façon simple.

Notez que dans la documentation de la librairie standard, ce module est classé dans la rubrique Generic Operating System Services (c'est-à-dire les services communs aux différents systèmes d'exploitation). Ce n'est pas un hasard : *time* est un module très proche du système. Cela signifie que certaines fonctions de ce module pourront avoir des résultats différents sous différents systèmes. Je vais pour ma part surtout m'attarder sur les fonctionnalités les plus génériques possibles, pour ne perdre personne .

Comme d'habitude, pour plus d'information, reportez-vous à la documentation du module time.

Représenter une date et une heure dans un nombre unique

Comment représenter un temps ? Il existe, naturellement, plusieurs réponses à cette question. Celle que nous allons voir ici est sans doute la moins compréhensible pour un humain, mais la plus adatpée pour un ordinateur : on stock la date et l'heure dans un seul entier.

Comment représenter une date et une heure dans un unique entier ?

L'idée retenue a été de représenter une date et une heure en fonction du nombre de secondes écoulées depuis une date précise. La plupart du temps, cette date est l'*Epoch Unix*, le 1 janvier 1970 à 00:00:00.

Pourquoi cette date plutôt qu'une autre ?

Il fallait bien choisir une date de début. L'année 1970 a été considérée comme un bon départ, sachant l'essort de l'informatique à partir de cette époque, et surtout sa généralisation. D'autre part, un ordinateur est inévitablement limité quand il traite des entiers et dans les langages de l'époque, il fallait tenir compte de ce fait tout simple : on ne pourrait pas compter le nombre de secondes jusqu'à une date très avancée. La date de l'*Epoch* ne pouvait donc pas être trop reculée dans le temps.

Nous allons voir dans un premier temps comment afficher ce fameux nombre de secondes écoulées depuis le 1 janvier 1970 à 00:00:00. On utilise la fonction *time* du module *time*.

Code : Python Console

```
>>> import time
>>> time.time()
1297642146.562
>>>
```

Ca fait beaucoup ! D'un autre côté, songez quand même que ça représente le nombre de secondes depuis plus de quarante ans à présent .

Maintenant, je vous l'accorde, ce nombre n'est pas très compréhensible pour un humain. Par contre, pour un ordinateur, c'est l'idéal : les durées calculées en nombre de secondes sont faciles à additionner, soustraire, multiplier... bref, l'ordinateur se débrouille bien mieux avec ce nombre de secondes, ce **timestamp** comme on l'appelle généralement.

Faites un petit test : stockez le retour de *time.time()* dans une première variable, puis quelques secondes plus tard stockez le

retour de <itaique>time.time()</italique> dans une autre variable. Comparez-les, soustrayez-les, vous verrez que ça se fait tout seul :

Code : Python Console

```
>>> debut = time.time()
>>> # On attend quelques secondes avant d'entrer la commande
suivante
... fin = time.time()
>>> print(debut, fin)
1297642195.45 1297642202.27
>>> debut < fin
True
>>> fin - debut # Combien de secondes entre debut et fin ?
6.812000036239624
>>>
```

Vous pouvez remarquer que le retour de *time.time()* n'est pas un entier mais bien un flottant. Le temps ainsi donné est plus précis qu'à une seconde près. Pour des calculs de performance, ce n'est en général pas cette fonction que l'on utilise. Mais c'est bien suffisant la plupart du temps .

La date et l'heure de façon plus présentable

Vous allez me dire que c'est bien joli d'avoir tous nos temps réduits à des nombres, mais que ce n'est pas très lisible pour nous. Nous allons découvrir tout au long du chapitre des moyens d'afficher nos temps de façon plus élégante et d'obtenir les diverses informations d'une date et d'une heure. Je vous propose ici un premier moyen : une sortie sous la forme d'un objet, contenant déjà beaucoup d'informations.

Nous allons utiliser la fonction *localtime* du module *time*.

Code : Python Console

```
time.localtime()
```

Elle retourne un objet contenant, dans l'ordre :

1. *tm_year* : l'année sous la forme d'un entier
2. *tm_mon* : le numéro du mois (entre 1 et 12)
3. *tm_mday* : le numéro du jour du mois (entre 1 et 31, variant d'un mois et d'une année à l'autre)
4. *tm_hour* : l'heure du jour (entre 0 et 23)
5. *tm_min* : le nombre de minutes (entre 0 et 59)
6. *tm_sec* : le nombre de secondes (entre 0 et 61, bien qu'on utilisera ici de 0 à 59, c'est bien suffisant)
7. *tm_wday* : un entier représentant le jour de la semaine (entre 0 et 6, 0 est lundi par défaut)
8. *tm_yday* : le jour de l'année, entre 1 et 366
9. *tm_isdst* : un entier représentant le changement d'heure local.

Comme toujours, si vous voulez en apprendre plus, je vous renvoie à la documentation officielle du module time.

Comme je l'ai dit plus haut, nous allons utiliser la fonction *localtime*. Elle prend un paramètre optionnel : le *timestamp* tel que nous l'avons découvert plus haut. Si ce paramètre n'est pas précisé, *localtime* utilisera automatiquement *time.time()* et retournera donc la date et l'heure actuelle.

Code : Python Console

```
>>> time.localtime()
time.struct_time(tm_year=2011, tm_mon=2, tm_mday=14, tm_hour=3,
```

```
tm_min=22, tm_se
c=7, tm_wday=0, tm_yday=45, tm_isdst=0)
>>> time.localtime(debut)
time.struct_time(tm_year=2011, tm_mon=2, tm_mday=14, tm_hour=1,
tm_min=9, tm_sec
=55, tm_wday=0, tm_yday=45, tm_isdst=0)
>>> time.localtime(fin)
time.struct_time(tm_year=2011, tm_mon=2, tm_mday=14, tm_hour=1,
tm_min=10, tm_se
c=2, tm_wday=0, tm_yday=45, tm_isdst=0)
>>>
```

Pour savoir à quoi correspond chaque attribut de l'objet, je vous renvoie un peu plus haut  . Pour l'essentiel, c'est assez clair je pense. Mais malgré tout, la date et l'heure retournées ne sont pas des plus lisibles. L'avantage de les avoir sous cette forme c'est qu'on peut facilement extraire une information, si on a, par exemple, juste besoin de l'année et du numéro du jour.

Récupérer un timestamp depuis une date

Je vais passer plus vite sur cette fonction, car selon toute vraisemblance vous l'utiliserez moins souvent. L'idée est, à partir d'une structure représentant une date et heure tel que retournées par *localtime* de récupérer le timestamp correspondant. On utilise pour ce faire la fonction *mktime*.

Code : Python Console

```
>>> print(debut)
1297642195.45
>>> temps = time.localtime(debut)
>>> print(temps)
time.struct_time(tm_year=2011, tm_mon=2, tm_mday=14, tm_hour=1,
tm_min=9, tm_sec
=55, tm_wday=0, tm_yday=45, tm_isdst=0)
>>> ts_debut = time.mktime(temps)
>>> print(ts_debut)
1297642195.0
>>>
```

Mettre en pause l'exécution du programme pendant un temps déterminé

C'est également une fonctionnalité intéressante, même si vous n'en voyez sans doute pas l'utilité de prime abord. La fonction qui nous intéresse est *sleep* et elle prend en paramètre un nombre de secondes qui peut être sous la forme d'un entier ou d'un flottant. Pour vous rendre compte de l'effet, je vous encourage à tester par vous-même  :

Code : Python Console

```
>>> time.sleep(3.5) # faire une pause pendant 3,5 secondes
>>>
```

Comme vous pouvez le voir, Python se met en pause et vous devez attendre 3,5 secondes avant que les trois chevrons s'affichent de nouveau.

Formatter un temps

Intéressons nous maintenant à la fonction *strftime*. Elle permet de formater une date et heure en la représentant dans une chaîne de caractère.

Elle prend deux paramètres :

- La chaîne de formatage (nous verrons plus bas comment la former)
- Un temps optionnel tel que le renvoie *localtime*. Si le temps n'est pas précisé, c'est la date et l'heure courante qui sont pris par défaut.

Pour construire notre chaîne de formatage, nous allons utiliser plusieurs caractères spéciaux. Python va remplacer ces caractères par leur valeur (une valeur du temps passé en second paramètre, ou du temps actuel sinon).

Exemple :

Code : Python Console

```
time.strftime('%Y')
```

Voici quelques symboles que vous pouvez utiliser dans cette chaîne :

Symbole	Signification
%A	Nom du jour de la semaine
%B	Nom du mois
%d	Jour du mois (de 01 à 31)
%H	Heure (de 00 à 23)
%M	Minute (entre 00 et 59)
%S	Seconde (de 00 à 59)
%Y	Année

Donc pour afficher la date tel qu'on y est habitué en France :

Code : Python Console

```
time.strftime("%A %d %B %Y %H:%M:%S")
```

Mais... c'est en anglais !

Et oui. Mais avec ce que vous savez déjà et ce que vous allez voir par la suite, vous n'aurez pas de difficultés à personnaliser tout ça  .

Bien d'autres fonctions

Le module *time* propose bien d'autres fonctions. Je ne vous ais montré que celles que j'utilise le plus souvent tout en vous présentant quelques concepts du temps utilisé en informatique. Si vous voulez aller plus loin, vous savez quoi faire... non ? Allez, je vous y encourage fortement donc je vous remet le lien vers la documentation du module *time* : http://docs.python.org/py3k/library/time.html.

Le module datetime

Le module *datetime* propose plusieurs classes pour représenter des dates et heures. Vous n'allez rien découvrir d'absolument spectaculaire dans cette sous-partie, mais nous nous avançons petit à petit vers une façon plus orientée objet de gérer des dates et heures.

Encore et toujours, je ne prétend pas remplacer la documentation. Je me contente d'extraire de celle-ci les informations qui me semblent les plus importantes. Je vous encourage, là encore, à jeter un oeil du côté de la documentation du module, que vous trouverez à l'adresse : http://docs.python.org/py3k/library/datetime.html.

Représenter une date

Vous le reconnaîtrez probablement avec moi, c'est bien d'avoir accès au temps actuel avec une précision d'une seconde sinon plus... mais parfois, cette précision est inutile. Dans certains cas, on a juste besoin d'une date, c'est-à-dire un jour, un mois et une année.

Il est naturellement possible d'extraire cette information de notre timestamp. Le module *datetime* propose une classe *date*, représentant une date, rien qu'une date  .

L'objet possède trois attributs :

- *year* : l'année
- *month* : le mois
- *day* : le jour du mois.

 Comment fait-on pour construire notre objet *date* ?

Il y a plusieurs façons de procéder. Le constructeur de cette classe prend trois arguments qui sont dans l'ordre l'année, le mois et le jour du mois.

Code : Python Console

```
>>> import datetime
>>> date = datetime.date(2010, 12, 25)
>>> print(date)
2010-12-25
>>>
```

Il existe deux méthodes de classe qui peuvent vous intéresser :

- *date.today()* : retourne la date d'aujourd'hui
- *date.fromtimestamp(timestamp)* : retourne la date correspondante au timestamp entré.</italique>

Voyons en pratique :

Code : Python Console

```
>>> import time
>>> import datetime
>>> aujourdhui = datetime.date.today()
>>> aujourdhui
datetime.date(2011, 2, 14)
>>> datetime.date.fromtimestamp(time.time()) # équivalent à
date.today
datetime.date(2011, 2, 14)
>>>
```

Et bien entendu, vous pouvez manipuler ces dates simplement et les comparer grâce aux opérateurs usuelles, je vous laisse

essayer !

Représenter une heure

C'est moins courant, mais on peut également être amené à manipuler une heure, indépendemment de toute date. La classe *time* du module *datetime* est là pour ça.

On construit une heure avec non pas trois mais cinq paramètres, tous optionnels :

- *hour* (0 par défaut) : l'heure, entre 0 et 23
- *minute* (0 par défaut) : la minute, entre 0 et 59
- *second* (0 par défaut) : la seconde, entre 0 et 59
- *microsecond* (0 par défaut) : la précision de l'heure en micro-secondes, entre 0 et 1.000.000
- *tzinfo* (*None* par défaut) : l'information de zone (je ne détaillerai pas cette information ici).

Cette classe est moins utilisée que *datetime.date* mais elle peut s'avérer utile dans certains cas. Je vous laisse faire quelques tests, n'oubliez pas de vous reporter à la documentation du module datetime pour plus d'informations.

Représenter des dates et heures

Et nous y voilà ! Vous n'allez pas être bien surpris par ce que nous allons voir. Nous avons vu une manière de représenter une date, une manière de représenter une heure, mais on peut naturellement représenter une date et une heure dans le même objet, ce sera probablement la classe que vous utiliserez le plus souvent. Celle qui nous intéresse s'appelle *datetime*... comme son module  .

Elle prend d'abord les paramètres de *datetime.date* (année, mois, jour) et ensuite les paramètres de *datetime.time* (heure, minute, seconde, micro-seconde et information sur la zone).

Voyons dès à présent les deux méthodes de classe que vous utiliserez le plus souvent :

- *datetime.now()* : retourne l'objet *datetime* avec la date et l'heure actuelle
- *datetime.fromtimestamp(timestamp)* : retourne la date et l'heure d'un timestamp précis.

Code : Python Console

```
>>> import datetime
>>> datetime.datetime.now()
datetime.datetime(2011, 2, 14, 5, 8, 22, 359000)
>>>
```

Il y a bien d'autres choses à voir dans ce module *datetime*. Si vous êtes curieux ou que vous avez des besoins plus spécifiques que je n'aborde pas ici, référez-vous à la documentation officielle du module, une fois de plus  .

Voir aussi :

- Documentation du module time
- Documentation du module datetime
- Documentation du module calendar

La gestion des temps est un peu plus complexe qu'il pourrait le sembler de prime abord. Mais vous avez à présent assez d'outils pour manipuler des temps dans vos programmes et surtout, vous savez comment aller plus loin si ce sujet vous intéresse particulièrement, ou que vous recherchez une fonctionnalité précise.

Dans le prochain chapitre, nous allons nous intéresser à d'autres concepts assez proches du système : nous parlerons entrées sorties standards, arguments de la ligne de commande, signaux et quelques autres concepts intéressants  .

Un peu de programmation système

Dans ce chapitre, nous allons découvrir plusieurs modules et fonctionnalités utiles pour interragir avec le système. Python peut servir à faire bien des choses, des jeux, des interfaces, mais il peut aussi faire des scripts systèmes et nous allons voir comment dans ce chapitre.

Les utilisateurs Linux risquent ici d'être plus familiers avec les concepts que je vais présenter. Pas de panique si vous êtes sous Windows toutefois : je vais prendre le temps de vous expliquer à chaque fois de ce dont on parle .

Les entrées et sorties standard

Pour commencer, nous allons voir comment accéder et manipuler les entrées et sorties standard.

Ca ressemble à quoi ?

Vous vous êtes sûrement habitué, quand vous utilisez la fonction *print*, à ce qu'un message s'affiche sur votre écran. Je pense que ça vous paraît assez logique, même.

Sauf que, comme d'ailleurs la plupart de nos manipulations en informatique, le mécanisme qui est derrière est plus complexe et puissant qu'il n'y paraît. Sachez que vous pourriez très bien faire en sorte qu'en utilisant *print*, ça écrive dans un fichier au lieu de l'afficher sur l'écran.

Quel intérêt ? print est fait pour afficher à l'écran non ?

Pas seulement, non. Mais nous verrons ça un peu plus loin. Disons pour l'instant que si, quand vous appelez la fonction *print*, le message s'affiche à l'écran, c'est parce que la sortie standard de votre programme est redirigée vers votre écran.

On distingue trois entrées et sorties standard :

- L'entrée standard : celle appelée quand vous utilisez *input*. C'est celle qui est utilisée pour demander des informations à l'utilisateur. Par défaut, l'entrée standard est votre clavier
- La sortie standard : comme on l'a vu, c'est celle qui est utilisée pour afficher des messages. Par défaut, elle redirige vers l'écran
- La sortie d'erreur : elle est notamment utilisée quand Python vous affiche le **traceback** d'une exception. Par défaut, elle redirige également vers votre écran.

Accéder aux entrées et sorties standard

On peut accéder aux objets représentant ces entrées et sorties standard grâce au module *sys* qui propose plusieurs fonctions et variables permettant d'interragir avec le système. Nous en reparlerons un peu plus tard dans le chapitre d'ailleurs .

Code : Python Console

```
>>> import sys
>>> sys.stdin # l'entr,e standard (standard input)
<_io.TextIOWrapper name='<stdin>' encoding='cp850'>
>>> sys.stdout # la sortie standard (standard output)
<_io.TextIOWrapper name='<stdout>' encoding='cp850'>
>>> sys.stderr # la sortie d'erreur (standard error)
<_io.TextIOWrapper name='<stderr>' encoding='cp850'>
>>>
```

Ces objets ne vous rappellent rien ? Non ?

Ils sont de la même classe que nos fichiers ouverts grâce à la fonction *open*. Et il n'y a aucun hasard derrière cela :

En effet, pour lire ou écrire sur nos entrées / sorties standard, on utilise les méthodes *read* et *write*.

Naturellement, l'entrée standard *stdin* peut lire (méthode *read*) et les deux sorties *stdout* et *stderr* peuvent écrire (méthode *write*).

Essayons quelque chose :

Code : Python Console

```
>>> sys.stdout.write("un test")
un test7
>>>
```

Pas trop de surprise, sauf que ce serai mieux avec un saut de ligne à la fin . Là, ce que retourne la méthode (le nombre de caractères écrits) est affiché juste après notre message .

Code : Python Console

```
>>> sys.stdout.write("Un test\n")
Un test
8
>>>
```

Modifier les entrées et sorties standard

Vous pouvez modifier *sys.stdin*, *sys.stdout* et *sys.stderr*. Faisons un premier test :

Code : Python Console

```
>>> fichier = open('sortie.txt', 'w')
>>> sys.stdout = fichier
>>> print("Quelque chose...")
>>>
```

Ici, rien ne s'affiche à l'écran. En revanche, si vous ouvrez le fichier *sortie.txt*, vous verrez le message que vous avez passé à *print*.

Je ne trouve pas le fichier sortie.txt, où est-il ?

Il doit se trouver dans le répertoire courant de Python. Pour connaître l'emplacement de ce répertoire, utilisez le module *os* et la fonction *getcwd* (Get Current Working Directory).

Une petite subtilité : si vous essayez de faire appel à *getcwd* directement, le résultat ne va pas s'afficher à l'écran... il va être écrit dans le fichier . Pour rétablir l'ancienne sortie standard, entrez la ligne :

Code : Python

```
sys.stdout = sys.__stdout__
```

Vous pouvez ensuite faire appel à la fonction *getcwd* :

Code : Python

```
import os
os.getcwd()
```

Dans ce répertoire, vous devriez trouver votre fichier *sortie.txt*.

Si vous avez modifié les entrées et sorties standard et que vous cherchez les objets d'origine, ceux redirigeant vers le clavier (pour l'entrée) et vers l'écran (pour les sorties), vous pouvez les trouver dans *sys.__stdin__*, *sys.__stdout__* et *sys.__stderr__*.

La documentation de Python nous conseille malgré tout de garder de préférence les objets d'origine sous la main plutôt que d'aller les chercher dans *sys.__stdin__*, *sys.__stdout__* et *sys.__stderr__*.

Voilà qui conclut notre bref aperçu des entrées et sorties standard. Là encore, si vous ne voyez pas d'application pratique à ce que je viens de vous montrer, peut-être cela viendra-t-il par la suite... ou pas .

Les signaux

Les signaux sont un des moyens dont dispose le système pour communiquer avec votre programme. Typiquement, si le système doit arrêter votre programme, il va lui envoyer un signal.

Les signaux peuvent être interceptés dans votre programme. Cela vous permet de faire une certaine action si le programme doit se fermer (enregistrer des objets en fichier, fermer les connexions réseau établies avec des clients éventuels, ...).

Les signaux sont également utilisés pour faire communiquer des programmes entre eux. Si votre programme est décomposé en plusieurs programmes s'exécutant indépendemment les uns des autres, cela permet de les synchroniser à certains moments clés. Nous ne verrons pas cette dernière fonctionnalité ici, il y aurai beaucoup de choses à aborder .

Les différents signaux

Le système dispose de plusieurs signaux génériques qu'il peut envoyer aux programmes quand cela est nécessaire. Si vous demandez l'arrêt du programme, un signal particulier lui sera envoyé.

Tous les signaux ne se retrouvent pas sous tous les systèmes d'exploitation, c'est pourquoi je vais surtout m'attacher à un signal : le signal *SIGINT* envoyé à l'arrêt du programme.

Pour plus d'informations, un petit détour par la documentation s'impose, notamment du côté du module signal que nous allons voir à présent.

Intercepter un signal

Commencez par importer le module *signal*.

Code : Python

```
import signal
```

Le signal qui nous intéresse, comme je l'ai dit, se nomme *SIGINT*.

Code : Python Console

```
>>> signal.SIGINT
2
>>>
```

Pour intercepter ce signal, il va falloir créer une fonction qui sera appelée si le signal est envoyé. Cette fonction prend deux paramètres :

- Le signal (plusieurs signaux peuvent être envoyés à la même fonction)
- Le **frame** qui ne nous intéresse pas ici.

Cette fonction, c'est à vous de la créer . Ensuite, il faudra la connecter avec notre signal *SIGINT*.

D'abord, créons notre fonction :

Code : Python

```
import sys

def fermer_programme(signal, frame):
    """Fonction appelée quand vient l'heure de fermer notre programme"""
    print("C'est l'heure de la fermeture !")
    sys.exit(0)
```

C'est quoi, la dernière ligne ?

On demande simplement à notre programme Python de se fermer. C'est le comportement standard quand on réceptionne un tel signal et notre programme doit bien s'arrêter à un moment où à un autre .

Pour ce faire, on utilise la fonction *exit* (sortir, en anglais) du module *sys*. Elle prend en paramètre le code de retour du programme.

Pour simplifier, la plupart du temps, si votre programme retourne *0*, le système comprendra que tout s'est bien passé. Si c'est un autre entier que *0*, le système interprétera ça comme une erreur ayant eu lieu pendant l'exécution de votre programme.

Ici, notre programme s'arrête normalement, on passe donc à *exit 0*.

Connectons notre fonction au signal *SIGINT* à présent. Sans quoi, notre fonction ne serai jamais appelée .

On utilise la fonction *signal* pour cela. Elle prend en paramètre :

- Le signal à intercepter
- La fonction que l'on doit connecter à ce signal.

Code : Python

```
signal.signal(signal.SIGINT, fermer_programme)
```

Ne mettez pas les parenthèses à la fin de la fonction . On envoie la référence vers la fonction, on ne l'exécute pas.

Cette ligne va connecter le signal *SIGINT* à la fonction *fermer_programme* que vous avez définie plus haut. Dès que le système enverra ce signal pour fermer le programme, la fonction *fermer_programme* sera appelée.

Pour vérifier que tout marche bien, lancez une boucle infinie dans votre programme :

Code : Python

```
print("Le programme va boucler...")
```

```
while True: # boucle infinie, True est toujours vrai
    continue
```

Je vous remet le code en entier, si cela vous est plus facile :

Code : Python

```
import signal
import sys

def fermer_programme(signal, frame):
    """Fonction appelée quand vient l'heure de fermer notre
programme"""
    print("C'est l'heure de la fermeture !")
    sys.exit(0)

# Connexion du signal à notre fonction
signal.signal(signal.SIGINT, fermer_programme)

# Notre programme...
print("Le programme va boucler...")
while True:
    continue
```

Quand vous lancez ce programme, vous voyez un message vous informant que le programme va boucler ... et le programme continue de tourner. Il ne s'arrête pas. Il ne fait rien, il boucle simplement, mais il va continuer de boucler tant que son exécution n'est pas arrêtée.

Dans la fenêtre du programme, entrez *CTRL* + *C* sous Windows ou Linux, *CTRL* + *C* sous Mac.

Cette combinaison de touche va demander au programme de s'arrêter. Après l'avoir fait, vous pouvez constater qu'effectivement, votre fonction *fermer_programme* est bien appelée et s'occupe de fermer le programme correctement .

Voilà pour les signaux. Si vous voulez aller plus loin, la documentation est par ici :
http://docs.python.org/py3k/library/signal.html.

Interpréter les arguments de la ligne de commande

Python nous offre plusieurs moyens, en fonction de nos besoins, pour interpréter les arguments de la ligne de commande. Pour faire court, ces arguments peuvent être des paramètres que vous passez au lancement de votre programme et qui influenceront son exécution.

Ceux travaillant sous Linux n'auront, je pense, aucun mal à me suivre. Mais je vais faire une petite présentation pour ceux qui viennent de Windows, afin qu'ils puissent suivre sans difficulté.

Si vous êtes alergique à la console, passez à la suite .

Accéder à la console sous Windows

Il existe plusieurs moyens d'accéder à la console sous Windows. Celui que j'utilise et que je vais vous montrer passe par le Menu Démarrer.

Entrez dans le Menu Démarrer, cliquez sur *exécuter...*. Dans la fenêtre qui s'affiche, tapez *cmd* puis appuyez sur Entrée.

Vous devriez vous retrouver dans une fenêtre en console, vous donnant plusieurs informations propres au système.

Code : Console

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [version 5.1.2600]
```

```
(C) Copyright 1985-2001 Microsoft Corp.
C:\Documents and Settings\utilisateur>
```

Ce qui nous intéresse, c'est la dernière ligne. C'est un chemin qui vous indique dans quel endroit de l'arborescence vous vous trouvez. Toutes les chances sont pour que ce chemin soit le répertoire utilisateur de votre compte.

Code : Console

```
C:\Documents and Settings\utilisateur>
```

Nous allons commencer par nous déplacer dans le répertoire contenant l'interpréteur Python. Là encore, si vous n'avez rien changé lors de l'installation de Python, c'est *C:\pythonXX*, les X représentants les deux premiers chiffres de votre version de Python. Sous Python 3.2, ce sera probablement donc *C:\python32*.

Déplacez-vous dans ce répertoire grâce à la commande *cd*.

Code : Console

```
C:\Documents and Settings\utilisateur>cd C:\python32
C:\Python32>
```

Si tout va bien, vous pouvez voir à la dernière ligne que vous êtes bien dans le répertoire Python.

En vérité, vous pouvez appeler Python de n'importe où dans l'arborescence, mais ce sera plus simple si l'on est dans le répertoire de Python pour commencer  .

Accéder aux arguments de la ligne de commande

Nous allons une fois encore faire appel à notre module *sys*. Cette fois, nous allons nous intéresser à sa variable *argv*.

Créez un nouveau fichier Python. Sous Windows, prenez bien soin de l'enregistrer dans le répertoire de Python (*C:\python32* sous Python 3.2).

Placez-y le code suivant :

Code : Python

```
import sys
print(sys.argv)
```

sys.argv contient une liste des arguments que vous passez en ligne de commande, au moment de lancer le programme. Essayez donc d'appeler votre programme dans la ligne de commande en lui passant des arguments.

Sous windows :

Code : Console

```
C:\Python32>python test_console.py
['test_console.py']
C:\Python32>python test_console.py arguments
['test_console.py', 'arguments']
C:\Python32>python test_console.py argument1 argument2 argument3
['test_console.py', 'argument1', 'argument2', 'argument3']
C:\Python32>
```

Comme vous le voyez, le premier élément de *sys.argv* contient le nom du programme, de la façon dont vous l'avez appelé. Le reste de la liste contient vos arguments (si il y en a).

Note : vous pouvez très bien avoir des arguments contenant des espaces. Pour cela, il vous faut entourer votre argument de guillemets :

Code : Console

```
C:\Python32>python test_console.py "un argument avec des espaces"
['test_console.py', 'un argument avec des espaces']
C:\Python32>
```

Interpréter les arguments de la ligne de commande

Accéder aux arguments, c'est bien, mais les interpréter peut être utile aussi.

Des actions simples

Parfois, votre programme devra faire plusieurs actions en fonction du premier paramètre entré. Vous pourriez préciser, en premier argument, *start* pour démarrer une opération, *stop* pour l'arrêter, *restart* pour la redémarrer, *status* pour connaître son statut... bref, les utilisateurs de Linux ont sûrement bien plus d'exemples à l'esprit  .

Dans ce cas de figure, il n'est pas vraiment nécessaire d'interpréter les arguments de la ligne de commande comme on va le voir. Notre programme Python ressemblerai simplement à ça :

Code : Python

```
import sys

if len(sys.argv) < 2:
    print("Précisez une action en paramètre")
    sys.exit(1)

action = sys.argv[1]

if action == "start":
    print("On démarre l'opération")
elif action == "stop":
    print("On arrête l'opération")
elif action == "restart":
    print("On redémarre l'opération")
elif action == "status":
    print("On affiche le statut (démarré ou arrêté ?) de
l'opération")
else:
    print("Je ne connais pas cette action")
```

Pour tester ce programme, il est nécessaire de passer par la ligne de commande. Pour ce faire sous Windows, vous avez les instructions au-dessus  .

Des options plus complexes

Mais la ligne de commande permet également de transmettre des arguments plus complexes, comme des options. La plupart du temps, nos options sont sous la forme : *-option_courte* (une seule lettre), *--option_longue*, suivi d'un argument ou non.

Souvent, une option courte est accessible depuis une option longue également.

Ici, mon exemple va être sous Linux (vous n'avez pas vraiment besoin d'être sous Linux pour le comprendre, rassurez-vous) :

La commande *ls* permet d'afficher le contenu d'un répertoire. On peut lui passer plusieurs options en paramètre qui influent sur ce que la commande va afficher au final.

Par exemple, pour afficher tous les fichiers (cachés ou non) du répertoire, on utilise l'option courte *a*.

Code : Console

```
$ ls -a
.  ..  fichier1.txt  .fichier_cache.txt  image.png
$
```

Cette option courte est accessible depuis une option longue, *all*. Vous arrivez donc au même résultat en entrant :

Code : Console

```
$ ls --all
.  ..  fichier1.txt  .fichier_cache.txt  image.png
$
```

Pour récapituler, nos options courtes sont précédées d'un seul tiret et composées d'une seule lettre. Les options longues sont précédées de deux tirets et sont composées de plusieurs lettres.

Certaines options attendent un argument, à préciser juste après l'option.

Par exemple, pour afficher les premières lignes d'un fichier, vous pouvez utiliser, toujours sous Linux, la commande *head*. Si vous voulez afficher les X premières lignes d'un fichier, vous utiliserez la commande *head -n X*.

Code : Console

```
$ head -n 5 fichier.txt
ligne 1
ligne 2
ligne 3
ligne 4
ligne 5
$
```

Dans ce cas, l'option *-n* attend un argument qui est le nombre de lignes à afficher.

Interpréter ces options grâce à Python

Cette petite présentation faite, revenons à Python .

Nous allons nous intéresser au module *getopt*, justement utile pour interpréter les arguments de la ligne de commande selon un certain schéma.

La fonction qui nous intéresse s'appelle *getopt*, comme son module . Elle prend en paramètre :

- La liste des arguments à interpréter. Dans notre cas, ce sera toujours *sys.argv[1:]* (on ne prend pas le premier paramètre qui est le nom du programme)
- Une chaîne représentant la suite des options courtes (d'une seule lettre donc)
- Une liste contenant plusieurs chaînes de caractères, représentant les options longues correspondantes aux options courtes. Elles doivent être entrées dans le même ordre que les options courtes. Ce paramètre est facultatif : les options

courtes peuvent très bien n'avoir aucune option longue correspondante.

Un exemple s'impose je pense :

Code : Python

```
import sys
import getopt
getopt.getopt(sys.argv[1:], "h", ["help"])
```

Cela signifie que notre argument de la ligne de commande peut éventuellement contenir une option, soit *-h* sous sa forme courte, soit *--help* sous sa forme longue.

Si vous voulez que vos options soient suivies de paramètres, faites suivre l'option courte d'un signe deux point *:* et l'option longue correspondante d'un signe égal =.

Code : Python

```
getopt.getopt(sys.argv[1:], "n:", ["number="])
```

Si une erreur se produit lors de l'interprétation des options, ce qui signifie que l'utilisateur n'a pas précisé les options comme elles devaient l'être, une exception *getopt.GetoptError* est levée. C'est pourquoi on met souvent notre appel à *getopt.getopt* dans un bloc *try*.

La fonction retourne deux informations : la liste des options, sous la forme d'une liste de couples (non verrons ça plus bas), et la liste des arguments non interprétés.

Je pense qu'un exemple avec tout le code d'interprétation sera plus parlant. Il est tiré de la documentation du module getopt :

Code : Python

```
import sys
import getopt

try:
    opts, args = getopt.getopt(sys.argv[1:], "ho:v", ["help",
"output="])
except getopt.GetoptError as err:
    # Affiche l'aide et quitte le programme
    print(err) # va afficher l'erreur en anglais
    usage() # fonction à écrire rappelant la syntaxe de la commande
    sys.exit(2)

output = None
verbose = False
for o, a in opts:
    if o == "-v":
        # On place l'option 'verbose' à True
        verbose = True
    elif o in ("-h", "--help"):
        # On affiche l'aide
        usage()
        sys.exit()
    elif o in ("-o", "--output"):
        output = a
    else:
        print("Option {} inconnue".format(o))
        sys.exit(2)
```

Vous pouvez passer à notre programme plusieurs options :

- *-h* ou *--help* : affichera l'aide de notre programme
- *-v* : activera l'option *verbose*, en général utilisée pour "faire parler" le programme de façon détaillée
- *-o* ou *--output*, cette option attend un paramètre.

Dans cet exemple, on ne traite pas les arguments supplémentaires passés à notre programme, mais si vous en avez besoin ils se trouveront dans votre variable *args*.

Si vous voulez en savoir plus, avoir d'autres exemples ou plus d'informations, je vous redirige naturellement vers la documentation du module getopt.

Si vous avez besoin de plus puissant, sachez qu'il existe, depuis Python 3.2 le module argparse qui remplace le module optparse. N'hésitez pas à y jeter un oeil .

Exécuter une commande système depuis Python

Nous allons ici nous intéresser à la façon d'exécuter des commandes depuis Python. Nous allons voir deux moyens, il en existe d'autres .

Ceux que je vais présenter fonctionneront sous Windows, même si les utilisateurs en auront moins l'utilité .

La fonction system

Vous vous souvenez peut-être de cette fonction du module *os*. Elle prend en paramètre une commande à exécuter, affiche le résultat de la commande et retourne son code de retour.

Code : Python

```
os.system("ls") # sous Linux
os.system("dir") # sous Windows
```

Vous pouvez capturer le code de retour de la commande, mais vous ne pouvez pas capturer le retour affiché par la commande.

En outre, la fonction *system* exécute un environnement particulier rien que pour votre commande. Cela veut dire, entre autre, que *system* retournera tout de suite même si la commande tourne toujours.

En gros, si vous faites *os.system("sleep 5")*, le programme ne s'arrêtera pas pendant cinq secondes.

La fonction popen

Cette fonction se trouve également dans le module *os*. Elle prend également en paramètre une commande.

Toutefois, au lieu de retourner le code de retour de la commande, elle retourne un objet, un pipe (un tuyau en français) qui vous permet de lire le retour de la commande.

Un exemple sous Linux :

Code : Python Console

```
>>> import os
>>> cmd = os.popen("ls")
>>> cmd
<os._wrap_close object at 0x7f81d16554d0>
>>> cmd.read()
'fichier1.txt\nimage.png\n'
>>>
```

Le fait de lire le pipe bloque le programme jusqu'à ce que la commande ait finie de s'exécuter.

Je vous ai dit qu'il existait d'autres moyens. Et au-delà de ça, vous avez beaucoup d'autres choses intéressantes dans le module *os* vous permettant d'interragir avec le système... et pour cause .

Une dernière fois (pour ce chapitre), je vous renvoie donc à la documentation, plus précisément à celle du module os.
Voilà, ce petit tour d'horizon est terminé. Ce n'était naturellement qu'un bref aperçu , c'est un vaste sujet, mais vous avez déjà de bonnes pistes pour avancer plus loin si vous en avez besoin .

Dans le prochain chapitre, nous ferons un peu de maths... rien de méchant vous allez voir !

Un peu de mathématiques

Dans ce chapitre, nous allons découvrir trois modules, certains que vous avez déjà utilisés à un moment ou à un autre .

- Le module *math* qui propose déjà un bon nombre de fonctions mathématiques
- Le module *fractions*, dont nous allons surtout voir la classe *Fraction*, permettant... vous l'avez deviné ? De modéliser des fractions
- Et enfin le module *random* que vous connaissez de par nos TP et que nous allons découvrir plus en détail ici.

Pour commencer, le module math

Le module *math*, vous le connaissez déjà : on l'a utilisé comme premier exemple de module créé par Python. Vous avez peut-être eu la curiosité de regarder l'aide du module pour voir quelles fonctions étaient définies. Dans tous les cas, je fais un petit point sur certaines de ces fonctions.

Je ne vais pas m'attarder très longtemps sur ce module en particulier, car il est plus vraissemblable que vous cherchiez une fonction précise et que la documentation sera, dans ce cas, plus accessible et explicite .

Fonctions usuelles

Vous vous souvenez des opérateurs +, -, *, / et % j'imagine, je ne vais peut-être pas y revenir .

Trois fonctions pour commencer notre petit tour d'horizon :

Code : Python Console

```
>>> math.pow(5, 2) # 5 au carré
25.0
>>> 5 ** 2 # pratiquement identique à pow(5, 2)
25
>>> math.sqrt(25) # racine carrée de 25 (square root)
5.0
>>> math.exp(5) # exponentiel
148.4131591025766
>>> math.fabs(-3) # valeur absolue
3.0
>>>
```

Il y a bel et bien une différence entre l'opérateur ** et *math.pow*. La fonction retourne toujours un flottant alors que l'opérateur retourne un entier quand cela est possible .

Un peu de trigonométrie

Avant de voir les fonctions usuelles en trigonométrie, j'attire votre attention sur le fait que les angles, en Python, sont donnés et retournés en radian (rad).

Pour rappel :

Citation

1 rad = 57,29 degré

Cela étant dit, il existe déjà dans le module *math* les fonctions qui vont nous permettre de convertir simplement nos angles .

Code : Python

```
math.degrees(angle en radian) # convertit en degré
```

```
math.radians(angle_en_degré) # convertit en radian
```

Cela étant posé, voyons les quelques fonctions. Elles se nomment sans surprise :

- *cos* : cosinus
- *sin* : sinus
- *tan* : tangente
- *acos* : arc cosinus
- *asin* : arc sinus
- *atan* : arc tangente

Arrondir un nombre

Le module *math* nous propose plusieurs fonctions pour arrondir un nombre selon différents critères :

Code : Python Console

```
>>> math.ceil(2.3) # retourne le plus petit entier >= 2.3
3
>>> math.floor(5.8) # retourne le plus grand entier <= 5.8
5
>>> math.trunc(9.5) # tronque 9.5
9
>>>
```

Quant aux constantes du module, elles ne sont pas nombreuses : *math.pi* naturellement, ainsi que *math.e*  .

Voilà, ce fut rapide mais suffisant, sauf si vous cherchez quelque chose de précis. Au quel cas, un petit tour du côté de la documentation officielle du module math s'impose.

Des fractions avec... le module fractions

Ce module propose, entre autre, de manipuler des objets modélisant des fractions. C'est la classe *Fraction* du module qui nous intéresse :

Code : Python

```
from fractions import Fraction
```

Créer une fraction

Le constructeur de la classe *Fraction* accepte plusieurs types de paramètres :

- Deux entiers, le numérateur et le dénominateur (par défaut le numérateur est *0* et le dénominateur est *1*). Si le dénominateur est *0*, lève une exception *ZeroDivisionError*
- Une autre fraction
- Une chaîne sous la forme *'numerateur/denominateur'*.

Code : Python Console

```
>>> un_demi = Fraction(1, 2)
>>> un_demi
```

```
Fraction(1, 2)
>>> un_quart = Fraction('1/4')
>>> un_quart
Fraction(1, 4)
>>> autre_fraction = Fraction(-5, 30)
>>> autre_fraction
Fraction(-1, 6)
>>>
```

Ne peut-on pas créer des fractions depuis un flottant ?

Si, mais pas dans le constructeur. Pour créer une fraction depuis un flottant, on utilise la méthode de classe *from_float* :

Code : Python

```
>>> Fraction.from_float(0.5)
Fraction(1, 2)
>>>
```

Et pour retomber sur un flottant, rien de plus simple  :

Code : Python Console

```
>>> float(un_quart)
0.25
>>>
```

Manipuler les fractions

Maintenant, quel intérêt d'avoir nos nombres sous cette forme ? Surtout pour la précision des calculs. Les fractions que nous venons de voir supportent naturellement les opérateurs usuels :

Code : Python Console

```
>>> un_dixieme = Fraction(1, 10)
>>> un_dixieme + un_dixieme + un_dixieme
Fraction(3, 10)
>>>
```

Alors que :

Code : Python Console

```
>>> 0.1 + 0.1 + 0.1
0.30000000000000004
>>>
```

Biensûr, la différence n'est pas énorme, mais elle est là. Tout dépend de vos besoins en terme de précision  .

D'autres calculs ?

Code : Python Console

```
>>> un_dixieme * un_quart
Fraction(1, 40)
>>> un_dixieme + 5
Fraction(51, 10)
>>> un_demi / un_quart
Fraction(2, 1)
>>> un_quart / un_demi
Fraction(1, 2)
>>>
```

Voilà. Une petite démonstration qui vous suffira si ce module vous intéresse. Et si elle ne suffit pas, rendez-vous sur la documentation officielle du module fractions, là encore .

Du pseudo-aléatoire avec random

Le module *random*, vous l'avez également utilisé, pendant nos TP. Nous allons voir quelques fonctions de ce module, le tour d'horizon sera rapide .

Du pseudo-aléatoire

L'ordinateur est une machine puissante, capable de faire beaucoup de choses. Mais lancer les dés n'est pas son fort. Une calculatrice standard n'a aucune difficulté à additionner, soustraire, multiplier ou diviser des nombres. Elle peut même faire des choses bien plus complexes. Mais choisir un nombre au hasard est bien plus compliqué qu'il n'y paraît, pour un ordinateur.

Ce qu'il faut bien comprendre, c'est que derrière notre appel à *random.randrange* par exemple, Pyton va faire un véritable calcul pour trouver un nombre aléatoire. De ce fait, le nombre généré n'est pas réellement aléatoire, puisqu'un calcul identique, dans les mêmes conditions, donnera le même nombre. Les algorithmes mis en place pour générer de l'aléatoire sont maintenant suffisamment complexes pour que les nombres générés ressemblent bien à une série aléatoire. Souvenez-vous toutefois que pour un ordinateur, le véritable hasard ne peut pas exister .

La fonction random

Cette fonction, on ne l'utilisera peut-être pas souvent directement, mais elle est implicitement utilisée par le module quand on fait appel à *randrange* ou *choice* que nous verrons plus bas.

Elle génère un nombre pseudo-aléatoire entre 0 et 1. Ce sera donc naturellement un flottant :

Code : Python Console

```
>>> import random
>>> random.random()
0.9565461152605507
>>>
```

randrange et randint

La fonction *randrange* prend trois paramètres :

- La marge inférieure de l'intervalle
- La marge supérieure de l'intervalle
- L'écart entre chaque valeur de l'intervalle (*1* par défaut).

Que veut dire le dernier paramètre ?

Prenons un exemple, ce sera plus simple :

Code : Python

```
random.randrange(5, 10, 2)
```

Cette instruction va chercher à générer un nombre aléatoire entre *5* inclus et *10* non inclus, avec un écart de *2* entre chaque valeur. Elle va donc chercher dans la liste des valeurs *[5, 7, 9]*.

Si vous ne précisez pas de troisième paramètre, il sera à *1* par défaut (c'est le comportement attendu la plupart du temps).

La fonction *randint* prend deux paramètres :

- La marge inférieure de l'intervalle là encore
- La marge supérieure de l'intervalle, cette fois inclus dedans.

Pour tirer au hasard un nombre entre *1* et *6*, il est donc plus intuitif de faire :

Code : Python

```
random.randint(1, 6)
```

Opérations sur des séquences

Nous allons voir deux fonctions : la première, *choice*, retourne au hasard un élément d'une séquence passée en paramètre :

Code : Python Console

```
>>> random.choice(['a', 'b', 'k', 'p', 'i', 'w', 'z'])
'k'
>>>
```

La seconde s'appelle *shuffle*. Elle prend une séquence en paramètre et la mélange ; elle modifie donc la séquence qu'on lui passe et ne retourne rien :

Code : Python Console

```
>>> liste = ['a', 'b', 'k', 'p', 'i', 'w', 'z']
>>> random.shuffle(liste)
>>> liste
['p', 'k', 'w', 'z', 'i', 'b', 'a']
>>>
```

Voilà. Là encore, ce fut rapide mais si vous voulez aller plus loin, vous savez ou aller... droit vers la documentation .
C'en est fini de ce chapitre. Dans le prochain, nous allons parler mot de passe, comment en récupérer et comment les chiffrer pour les protéger .

Gestion des mots de passe

Dans ce chapitre, nous allons nous intéresser aux mots de passe et à la façon de les gérer en Python. D'abord de les réceptionner, ensuite de les protéger .

Nous allons découvrir deux modules dans ce chapitre : d'abord *getpass* qui permet de demander un mot de passe à l'utilisateur, puis *hashlib* qui permet de chiffrer le mot de passe réceptionné.

Réceptionner un mot de passe entré par l'utilisateur

Vous allez me dire, j'en suis sûr, qu'on a déjà une façon de réceptionner une entrée de l'utilisateur. Cette façon, on l'a vue assez tôt dans le cours : il s'agit naturellement de la fonction *input*.

Mais *input* n'est pas très discrète. Si vous entrez un mot de passe confidentiel, il apparaît visible à l'écran, ce qui n'est pas toujours souhaitable. Quand on entre un mot de passe c'est même rarement souhaité .

C'est ici qu'intervient le module *getpass*. La fonction qui nous intéresse porte le même nom que le module. Elle va réagir comme *input*, attendre une entrée de l'utilisateur et la retourner. Mais à la différence d'*input*, elle ne va pas afficher ce que l'utilisateur entre.

Faisons un essai :

Code : Python Console

```
>>> from getpass import getpass
>>> mot_de_passe = getpass()
Password:
>>> mot_de_passe
'un mot de passe'
>>>
```

Comme vous le voyez... bah justement on ne voit rien , rien du mot de passe que l'on entre. Vous appuyez sur les touches de votre clavier mais rien ne s'affiche. Vous écrivez bel et bien cependant et quand vous appuyez sur *Entrée*, la fonction *getpass* retourne ce que vous avez entré.

Ici, on le stock dans la variable *mot_de_passe*. C'est plus discret qu'*input*, reconnaissez-le .

Bon, il reste un détail, mineur certes mais un détail quand même : le **prompt** par défaut, c'est-à-dire le message qui vous invite à entrer votre mot de passe, est en anglais. Heureusement, il s'agit tout simplement d'un paramètre facultatif de la fonction :

Code : Python Console

```
>>> mot_de_passe = getpass("Entrez votre mot de passe : ")
Entrez votre mot de passe :
>>>
```

C'est mieux.

Bien entendu, tous les mots de passe que vous réceptionnerez ne viendront pas forcément d'une entrée directe d'un utilisateur. Mais dans ce cas précis, la fonction *getpass* est bien utile. A la fin de ce chapitre, nous verrons une utilisation complète de cette fonction, réception et chiffrement de notre mot de passe en prime, deux en un .

Chiffrer un mot de passe

Cette fois-ci, nous allons nous intéresser au module *hashlib*. Mais avant de vous montrer comment il fonctionne, quelques explications s'imposent.

Chiffrer un mot de passe ?

La première question qu'on pourrait légitimement se poser est "pourquoi protéger un mot de passe". Je suis sûr que vous pouvez trouver par vous-mêmes pas mal de réponses : il est un peu trop facile de récupérer un mot de passe si il est stocké ou

transmis en clair. Et avec un mot de passe, on peut avoir accès à beaucoup de choses je n'ai pas besoin de vous l'expliquer . Ce qui fait que généralement, quand on a besoin de stocker un mot de passe ou de le transmettre, on le chiffre.

Maintenant, qu'est-ce que le chiffrement ? A priori, l'idée est assez simple : en partant d'un mot de passe, n'importe lequel, on arrive à une seconde chaîne de caractère, complètement incompréhensible .

Quel intérêt ?

Et bien, si vous voyez passer, devant vos yeux, un truc comme `b47ea832576a75814e13351dcc97eaa985b9c6b7`, vous ne pouvez pas vraiment deviner le mot de passe qui se cache derrière.

Et l'ordinateur ne peut pas le déchiffrer si facilement que ça, non plus. Biensûr, il existe des méthodes pour déchiffrer un mot de passe, mais nous ne les verrons certainement pas ici . Nous, ce qu'on veut savoir, c'est comment protéger nos mots de passe, pas comment déchiffrer ceux des autres .

Comment marche le chiffrement ?

Grave question. D'abord, il y a plusieurs techniques de chiffrement, plusieurs **algorithmes** de chiffrement. Chiffrer un mot de passe avec un algorithme ne donne pas le même résultat qu'avec un autre algorithme.

Ensuite, l'algorithme, quel qu'il soit, est assez complexe. Je serai bien incapable de vous expliquer en détail comment ça marche, on fait appel à beaucoup de concepts mathématiques relativement poussés.

Mais si vous voulez faire un exercice, je vous propose un petit quelque chose amusant qui vous donnera une meilleure idée du chiffrement, même si naturellement en réalité c'est bien plus complexe .

Commencez par numéroter toutes les lettres de l'alphabet (de *a* à *z*) de *1* à *26*. Représentez chaque valeur dans un tableau, ce sera plus simple .

A (1)	B (2)	C (3)	D (4)	E (5)	F (6)	
G (7)	H (8)	I (9)	J (10)	K (11)	L (12)	M (13)
N (14)	O (15)	P (16)	Q (17)	R (18)	S (19)	
T (20)	U (21)	V (22)	W (23)	X (24)	Y (25)	Z (26)

Maintenant, disons qu'on va chercher à chiffrer des prénoms. Pour ça, on va prendre pour exemple un calcul simple : prenez chaque lettre constituant le prénom et ajoutez leur valeur respective selon le tableau ci-dessus.

Par exemple, on a le prénom *Eric*. Quatre lettres, ça ira vite . Oubliez les accents, les majuscules et minuscules. On a un *E (5)*, un *R (18)*, un *I (9)* et un *C (3)*. En ajoutant les valeurs de chaque lettre, on a donc *5 + 18 + 9 + 3*. Ce qui donne *35*.

Conclusion : en chiffrant *Eric* grâce à notre algorithme, on obtient le nombre *35*.

C'est l'idée derrière le chiffrement, même si en réalité c'est bien bien plus complexe. En outre, au lieu d'avoir un chiffre en sortie, on a plutôt une chaîne de caractères .

Mais prenez cet exemple pour vous amuser, si vous voulez. Appliquez notre algorithme à plusieurs prénoms. Si vous vous sentez d'attaque, essayez de faire une fonction Python qui prenne en paramètre notre chaîne et retourne un chiffre, ce n'est pas bien difficile .

Vous pouvez maintenant vous rendre compte que derrière un nombre tel que *35*, il est plutôt difficile de deviner que se cache le prénom *Eric* .

Si vous faites le test sur le prénom *Louis* et *Jacques*, vous vous rendrez compte... qu'ils ont le même nombre *76*. En effet :

- *Louis = 12 + 15 + 21 + 9 + 19 = 76*
- *Jacques = 10 + 1 + 3 + 17 + 21 + 5 + 19 = 76*

C'est ce qu'on appelle une **collision** : en prenant deux chaînes différentes, on obtient le même chiffrement au final.

Les algorithmes que nous allons voir dans le module *hashlib* essayent de minimiser au possible les collisions. Celui que nous venons juste de voir en est plein : il suffit de changer de place les lettres de notre prénom et on retombe sur le même nombre, après tout  .

Voilà. Fin de l'exercice, on va se pencher sur notre module *hashlib* maintenant  .

Chiffrer un mot de passe

On peut comencer par importer le module *hashlib*  :

Code : Python

```
import hashlib
```

On va maintenant choisir un algorithme. Pour nous aider dans notre choix, le module *hashlib* nous propose deux listes :

- *algorithms_guaranteed* : les algorithmes garantis par Python, les mêmes d'une plateforme à l'autre. Si vous voulez faire des programmes portables, il est préférable d'utiliser un de ces algorithmes :

Code : Python Console

```
>>> hashlib.algorithms_guaranteed
{'sha1', 'sha224', 'sha384', 'sha256', 'sha512', 'md5'}
>>>
```

- *algorithms_available* : les algorithmes disponibles sur votre plateforme. Tous les algorithmes garantis s'y trouvent, plus quelques autres propres à votre système.

Nous allons nous intéresser à *sha1* dans ce chapitre.

Pour commencer, nous allons créer notre objet *SHA1*. On va utiliser le constructeur *sha1* du module *hashlib*. Il prend en paramètre une chaîne, mais une chaîne de *bytes*.

Pour obtenir une chaîne de *bytes* depuis une chaîne *str*, on peut utiliser la méthode *encode*. Je ne vais pas rentrer dans le détail des encodages ici. Vous avez une autre possibilité, pour écrire directement une chaîne *bytes* sans passer par une chaîne *str*, c'est de mettre un *b* minuscule avant l'ouverture de votre chaîne :

Code : Python Console

```
>>> b'test'
b'test'
>>>
```

Générons notre mot de passe :

Code : Python Console

```
>>> mot_de_passe = hashlib.sha1(b"mot de passe")
>>> mot_de_passe
<sha1 HASH object @ 0x00BF0ED0>
>>>
```

Pour obtenir le chiffrement associé à cet objet, on a deux possibilités :

- la méthode *digest* qui retourne un type *bytes* contenant notre mot de passe chiffré
- La méthode *hexdigest* retournant une chaîne *str* contenant une suite de symboles hexadécimaux (de 0 à 9 plus de A à F).

C'est cette dernière méthode que je vais montrer ici, parce qu'elle est préférable pour un stockage en fichier si les fichiers doivent transiter d'une plateforme à l'autre.

Code : Python Console

```
>>> mot_de_passe.hexdigest()
'b47ea832576a75814e13351dcc97eaa985b9c6b7'
>>>
```

Et pour décrypter ce mot de passe ?

On ne le décrypte pas. On ne le déchiffre pas  . Si vous voulez savoir si le mot de passe entré par l'utilisateur correspond au chiffrement que vous avez conservé, chiffrez de nouveau le mot de passe et comparez les deux chiffrements obtenus :

Code : Python

```
import hashlib
from getpass import getpass

chaine_mot_de_passe = b"azerty"
mot_de_passe_chiffre = hashlib.sha1(chaine_mot_de_passe).hexdigest()

verrouille = True
while verrouille:
    entre = getpass("Entrez le mot de passe : ") # azerty
    # On encode l'entrée pour avoir un type bytes
    entre = entre.encode()

    entre_chiffre = hashlib.sha1(entre).hexdigest()
    if entre_chiffre == mot_de_passe_chiffre:
        verrouille = False
    else:
        print("Mot de passe incorrect")

print("Mot de passe accepté...")
```

Voilà pour résumer tout le chapitre  . Ca me semble assez clair. Nous avons utilisez l'algorithme *sha1*, il en existe d'autres comme vous pouvez le voir dans *hashlib.algorithms_available*.

Je m'arrête pour ma part ici  . Si vous voulez aller plus loin...

- Documentation du module getpass
- Documentation du module hashlib

Voilà pour les mots de passe. Dans le prochain chapitre, nous allons aborder un sujet que vous trouverez sans doute des plus intéressants : le réseau !

Le réseau

Vaste sujet que le réseau ! Si je devais faire une présentation détaillée, ou même parler des réseaux en général, il me faudrait bien plus d'un chapitre ne serai-ce que pour la théorie .

Dans ce chapitre, nous allons donc apprendre à faire communiquer deux applications grâce aux **sockets**, des objets qui permettent de connecter un client et un serveur et de transmettre des données de l'un à l'autre.

Si ça vous met pas l'eau à la bouche...

Brève présentation du réseau

Comme je l'ai dit plus haut, le réseau est un sujet bien trop vaste pour que je le présente en un unique chapitre. On va s'attacher ici à comprendre comment faire communiquer deux applications, qui peuvent être sur la même machine mais aussi sur des machines distantes. Dans ce cas, elles se connectent grâce au réseau local ou à Internet.

Il existe plusieurs protocoles de communication en réseau. Si vous voulez, c'est un peu comme une langue : pour que les échanges se passent correctement, les deux (ou plus) parties en présence doivent parler la même . Nous allons ici parler du protocole TCP.

Le protocole TCP

Ce protocole signifie Transmission Control Protocol, soit "protocole de contrôle de transmission". Concrètement, il permet de connecter deux applications et de les faire échanger des informations.

Ce protocole est dit orienté connexion, c'est-à-dire que les applications sont connectées pour échanger et que l'on peut être sûr, quand on envoie une information au travers du réseau, qu'elle a bien été réceptionnée par l'autre application. Si la connexion est rompue, pour X raison, les applications doivent rétablir la connexion pour communiquer de nouveau.

Cela vous paraît peut-être évident, mais le protocole UDP par exemple (User Datagram Protocol) envoie des informations au travers du réseau sans se soucier de savoir si elles seront bien réceptionnées par la cible. Ce protocole n'est pas connecté, une application envoie quelque chose au travers du réseau en spécifiant une cible. Il suffit alors de prier très fort pour que le message soit réceptionné correctement .

Plus sérieusement, ce type de protocole est utile si vous avez besoin de faire transiter beaucoup d'informations au travers du réseau mais qu'une petite perte occasionnelle d'informations n'est pas très handicapante. On trouve ce type de protocole dans des jeux graphiques en réseau, le serveur envoyant très fréquemment des informations au client pour qu'il actualise sa fenêtre. Ca fait beaucoup à transmettre, mais ce n'est pas dramatique si il y a une petite perte d'informations de temps à autre puisque quelques milisecondes plus tard, le serveur renverra de nouveau les informations.

En attendant, c'est le protocole TCP qui nous intéresse. Il est un peu plus lent que le protocole UDP, mais plus sûr, et pour la quantité d'informations que nous allons transmettre, il est préférable d'être sûr des informations transmises plutôt que de la vitesse de transmission.

Clients et serveur

Dans l'architecture que nous allons voir dans ce chapitre, en général on trouve un serveur et plusieurs clients. Le serveur, c'est une machine qui va traiter les requêtes du client.

Si vous accédez au site du zéro, c'est parce que votre navigateur, faisant office de client, se connecte au serveur du site du zéro. Il lui envoie un message en lui demandant la page que vous souhaitez afficher et le serveur du site du zéro, dans sa grande bonté, envoie la page demandée au client.

Ce type d'architecture est très fréquente, même si ce n'est pas la seule envisageable.

Dans les exemples que nous allons voir, nous allons créer deux applications : l'application **serveur** et l'application **client**. Le serveur **écoute** donc en attendant des connexions et les clients se connectent au serveur.

Les différentes étapes

Nos applications vont fonctionner selon un schéma assez analogue. Voici dans l'ordre les étapes du client et du serveur. Les étapes sont très simplifiées, la plupart des serveurs peuvent communiquer avec plusieurs clients mais nous ne verrons pas ça tout de suite. Avant la fin de ce chapitre, en revanche .

Le serveur :

- Attend une connexion de la part du client
- Accepte la connexion quand le client se connecte
- Echange des informations avec le client
- Ferme la connexion.

Le client :

- Se connecte au serveur
- Echange des informations avec le serveur
- Ferme la connexion.

Comme on l'a vu, le serveur peut dialoguer avec plusieurs clients. C'est tout l'intérêt. Si le serveur du site du zéro ne pouvait dialoguer qu'avec un seul client à la fois, il faudrait attendre votre tour, peut-être assez longtemps, avant d'avoir accès à vos pages . Et sans serveur pouvant dialoguer avec plusieurs clients, les jeux en réseau, les logiciels de messagerie instantanés, seraient bien plus complexes .

Etablir une connexion

Pour que le client se connecte au serveur, il nous faut deux informations :

- Le **nom d'hôte** (**host name** en anglais) qui identifie une machine sur Internet ou sur un réseau local. Les noms d'hôte permettent de représenter des adresses IP de façon plus claire (on a un nom comme *google.fr*, plus facile à retenir que l'adresse IP correspondante *74.125.224.84*)
- Un numéro de port qui est souvent propre au type d'information que l'on va échanger. Si on demande une connexion Web, le navigateur va souvent interroger le port *80* si c'est en **http**, ou le port *443* si c'est en connexion sécurisée (**https**). Le numéro de port est compris entre 0 et 65535 (il y en a donc un certain nombre) et les numéros entre 0 et 1023 sont réservés par le système. On peut les utiliser, mais ce n'est pas une super bonne idée .

Pour résumer, quand votre navigateur tente d'accéder au site http://www.siteduzero.com, il établit une connexion avec le serveur dont le nom d'hôte est **siteduzero.com** sur le port *80*. Nous allons plus volontier travailler avec des noms d'hôte dans ce chapitre qu'avec des adresses IP.

Les sockets

Comme on va le voir, les **socket**s sont des objets qui permettent d'ouvrir une connexion avec une machine, locale ou distante et d'échanger avec elle.

Ces objets sont définis dans le module **socket** et nous allons maintenant voir comment ils fonctionnent .

Les sockets

Commençons donc, dans la joie et la bonne humeur , par importer notre module *socket*.

Code : Python

```
import socket
```

Nous allons d'abord créer notre serveur, puis en parallèle, un client. Nous allons faire communiquer les deux. Pour l'instant, nous nous occupons du serveur.

Construire notre socket

Nous allons pour cela faire appel au constructeur *socket*. Dans le cas d'une connexion **TCP**, il prend les deux paramètres suivants, dans l'ordre :

- *socket.AF_INET* : la famille d'adresse, ici ce sont des adresses Internet
- *socket.SOCK_STREAM* : le type du socket, *SOCK_STREAM* pour le protocole TCP.

Code : Python Console

```
>>> connexion_principale = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
>>>
```

Connecter le socket

Ensuite, on connecte notre socket. Pour une connexion serveur, qui va attendre des connexions de clients, on utilise la méthode *bind*. Elle prend un paramètre : le tuple *(nom_hote, port)*.

Attends un peu, je croyais que c'était notre client qui se connectait à notre serveur, pas l'inverse...

Oui, mais pour que notre serveur écoute sur un port, il faut le configurer pour  . Donc le nom de l'hôte sera vide dans notre cas, et le port sera celui que vous voulez, entre 1024 et 65535.

Code : Python Console

```
>>> connexion_principale.bind(('', 12800))
>>>
```

Faire écouter notre socket

Bien. Notre socket est prêt à écouter sur le port *12800*, mais il n'écoute pas encore  . On va avant tout lui préciser combien de connexions maximum il peut recevoir sur ce port, sans les accepter. On utilise pour cela la méthode *listen*. On lui passe généralement *5* en paramètre.

Ca veut dire que notre serveur ne pourra dialoguer qu'avec 5 clients maximum ?

Non. Ca veut dire que si 5 clients se connectent et que le serveur n'accepte aucune des connexions, aucun client de plus ne pourra se connecter. Mais généralement, très peu de temps après que le client ait demandé la connexion, le serveur l'accepte. Vous pouvez donc avoir bien plus de clients connectés  ne vous en faites pas.

Code : Python

```
>>> connexion_principale.listen(5)
>>>
```

Accepter une connexion venant du client

Et enfin, dernière étape, on va accepter une connexion. Aucune connexion ne s'est encore présentée, mais la méthode *accept* que nous allons utiliser va bloquer le programme tant qu'aucun client ne s'est connecté.

Une chose importante : la méthode *accept* retourne deux informations :

- Le socket connecté qui vient de se créer, celui qui va nous permettre de dialoguer avec notre client tout juste connecté
- Un tuple représentant l'adresse IP et le port de connexion du client.

Le port de connexion du client... est-ce que c'est pas le même que celui du serveur ?

Non, car votre client, en ouvrant une connexion, passe par un port, dit de sortie, qui va être choisi par le système dans les ports disponibles. Pour schématiser, quand un client se connecte à un serveur, il emprunte un port (une forme de porte, si vous voulez) puis établit le connexion sur le port du serveur. Il y a donc deux ports dans notre histoire, mais celui qu'utilise le client pour ouvrir sa connexion ne va pas nous intéresser.

Code : Python Console

```
>>> connexion_avec_client, infos_connexion =
connexion_principale.accept()
```

Cette méthode, comme vous le voyez, bloque le programme. Elle attend qu'un client se connecte. Laissons cette fenêtre Python ouverte et ouvrons-nous en une nouvelle pour construire notre client à présent.

Création du client

Commencez par construire votre socket de la même façon :

Code : Python Console

```
>>> import socket
>>> connexion_avec_serveur = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
>>>
```

Connecter le client

Pour se connecter à un serveur, on va utiliser la méthode *connect*. Elle prend en paramètre un tuple, comme *bind*, contenant le nom d'hôte et le numéro du port identifiant le serveur auquel on veut se connecter.

Le numéro du port sur lequel on veut se connecter, vous le connaissez : c'est *12800*. Vu que nos deux applications Python sont sur la même machine, le nom d'hôte va être *localhost* (c'est-à-dire la machine locale).

Code : Python Console

```
>>> connexion_avec_serveur.connect(('localhost', 12800))
>>>
```

Et voilà, notre serveur et notre client sont connectés !

Si vous retournez dans la console Python abritant le serveur, vous pouvez constater que la méthode *accept* ne bloque plus,

puisqu'elle vient d'accepter la connexion demandée par le client. Vous pouvez donc de nouveau entrer du code côté serveur :

Code : Python Console

```
>>> print(infos_connexion)
('127.0.0.1', 2901)
>>>
```

La première information, c'est l'adresse IP du client, ici *127.0.0.1* c'est-à-dire l'IP de l'ordinateur local. Dites-vous que l'hôte *localhost* redirige vers l'IP *127.0.0.1*.

Le second est le port de sortie du client, qui ne nous intéresse pas ici.

Faire communiquer nos sockets

Bon, maintenant comment faire communiquer nos sockets ? Et bien en utilisant les méthodes *send* pour envoyer et *recv* pour recevoir (receive).

Les informations que vous transmetterez seront des chaînes bytes, pas des str.

Donc côté serveur :

Code : Python Console

```
>>> connexion_avec_client.send(b"Je viens d'accepter la connexion")
32
>>>
```

La méthode *send* vous retourne le nombre de caractères envoyés.

Maintenant, côté client, on va réceptionner le message que l'on vient d'envoyer. La méthode *recv* prend en paramètre le nombre de caractères à lire. Généralement, on lui passe *1024*. Si le message est plus grand que 1024 caractères, on récupérera le reste après  .

Dans la fenêtre Python côté client, donc :

Code : Python Console

```
>>> msg_recu = connexion_avec_serveur.recv(1024)
>>> msg_recu
b"Je viens d'accepter la connexion"
>>>
```

Magique, non ?  vraiment pas ? Songez que ce petit mécanisme peut servir à faire communiquer non seulement des applications entre elles, mais aussi que ces applications peuvent se trouver sur des machines distantes et reliées par Internet 
.

Le client peut également envoyer des informations au serveur et le serveur peut les réceptionner, tout ça grâce aux méthodes *send* et *recv* que nous venons de voir.

Fermer la connexion

Pour fermer la connexion, il faut appeler la méthode *close* de notre socket.

Côté serveur :

Code : Python Console

```
>>> connexion_avec_client.close()
>>>
```

Et côté client :

Code : Python Console

```
>>> connexion_avec_serveur.close()
>>>
```

Voilà . Je vais récapituler en vous présentant dans l'ordre un petit serveur et un petit client que nous pouvons utiliser. Et pour finir, je vous montrerai une façon d'optimiser un peu notre serveur en lui permettant de gérer plusieurs clients à la fois .

Le serveur

Pour éviter les confusions, je vous remet ici le code du serveur, légèrement amélioré. Il n'accepte qu'un seul client (nous verrons plus bas comment en accepter plusieurs) et il tourne jusqu'à recevoir du client le message *fin*.

A chaque fois que le serveur reçoit un message, il envoie en retour le message *'5 / 5'*.

Code : Python

```
import socket

hote = ''
port = 12800

connexion_principale = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
connexion_principale.bind((hote, port))
connexion_principale.listen(5)
print("Le serveur écoute à présent sur le port {}".format(port))

connexion_avec_client, infos_connexion =
connexion_principale.accept()

msg_recu = b""
while msg_recu != b"fin":
    msg_recu = connexion_avec_client.recv(1024)
    # L'instruction ci-dessous peut lever une exception si le
message
    # réceptionné comporte des accents
    print(msg_recu.decode())
    connexion_avec_client.send(b"5 / 5")

print("Fermetures de connexions")
connexion_avec_client.close()
connexion_principale.close()
```

Voilà pour le serveur. Il est minimale vous en conviendrez mais il est fonctionnel . Nous verrons un peu plus loin comment l'améliorer.

Le client

Là encore, je vous propose le code du client pouvant interragir avec notre serveur.

Il va tenter de se connecter sur le port *12800* de la machine locale. Il demande à l'utilisateur d'entrer quelque chose au clavier et envoie ce quelque chose au serveur, puis attend sa réponse.

Code : Python

```
import socket

hote = "localhost"
port = 12800

connexion_avec_serveur = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
connexion_avec_serveur.connect((hote, port))
print("Connexion établie avec le serveur sur le port
{}".format(port))

msg_a_envoyer = b""
while msg_a_envoyer != b"fin":
    msg_a_envoyer = input("> ")
    # Peut planter si vous entrez des caractères spéciaux
    msg_a_envoyer = msg_a_envoyer.encode()
    # On envoie le message
    connexion_avec_serveur.send(msg_a_envoyer)
    msg_recu = connexion_avec_serveur.recv(1024)
    print(msg_recu.decode()) # là encore, peut planter si y'a des
accents

print("Fermeture de la connexion")
connexion_avec_serveur.close()
```

Que font les méthodes encode et decode ?

encode est une méthode de *str*. Elle peut prendre en paramètre un nom d'encodage et permet de passer un *str* en chaîne *bytes*. C'est, comme vous le savez, ce type de chaîne que *send* accepte. En fait, *encode* **encode** la chaîne *str* en fonction d'un encodage précis (par défaut, *Utf-8*).

decode, à l'inverse, est une méthode de *bytes*. Elle peut prendre aussi en paramètre un encodage et retourne une chaîne *str* décodée grâce à l'encodage (par défaut *Utf-8*).

Si l'encodage de vore console est différent d'*Utf-8* (ce sera souvent le cas sous Windows), des erreurs peuvent se produire si des accents se trouvent dans les messages que vous encodés ou décodés.

Voilà, nous avons vu un serveur et un client, tous deux très simples. Maintenant, voyons quelque chose de plus élaboré  .

Un serveur plus élaboré

Quel sont les problèmes de notre serveur ?

En vous posant la question, vous pouvez en trouver un certain nombre :

- D'abord, notre serveur ne peut accepter qu'un seul client. Si d'autres clients veulent se connecter, ils ne peuvent pas 
- Ensuite, on part toujours du principe qu'on attend le message d'un client et qu'on lui renvoie tout de suite quelque chose après. Mais ce n'est pas toujours le cas : parfois vous envoyez un message au client alors qu'il ne vous a rien envoyé, parfois vous recevez des informations de sa part alors que vous ne lui avez rien envoyé  .

  Prenez un logiciel de messagerie instantanée : est-ce que pour dialoguer vous êtes obligé d'attendre que celui avec lequel vous parlez vous réponde ? Ce n'est pas "j'envoie un message, il me répond, je lui répond, il me répond"... Parfois, souvent même, vous enverrez deux messages à la suite, trois peut-être même, ou l'inverse qui sait ?

Bref, on doit pouvoir envoyer plusieurs messages à notre client et réceptionner plusieurs messages dans un ordre inconnu. Avec notre technique, c'est impossible (faites le test si vous voulez).

En outre, les erreurs sont assez mal gérées vous en conviendrez .

Le module select

Le module *select* va nous permettre une chose très intéressante, être capable d'interroger plusieurs clients en l'attente d'un message à réceptionné, sans paralyser notre programme. Nous n'allons pas ici faire de la programmation parallèle, avec plusieurs morceaux de notre programme s'exécutant au même moment.

Pour schématiser, *select* va écouter sur une liste de clients et retourner au bout d'un temps précisé. Ce que retourne *select*, c'est la liste des clients qui ont un message à réceptionner. Il suffit de parcourir ces clients, lire les messages en attente (grâce à *recv*) et le tour est joué.

Sous Linux, *select* peut être utilisé sur autre chose que des sockets, mais cette fonctionnalité n'étant pas portable, je ne fais que la mentionner ici .

En théorie

La fonction qui nous intéresse porte le même nom que son module, *select*. Elle prend trois à quatre arguments et en retourne trois. C'est maintenant qu'il faut être attentif :

Les arguments que prend la fonction sont :

- *rlist* : la liste des sockets en attente d'être lu
- *wlist* : la liste des sockets en attente d'être écrit
- *xlist* : la liste des sockets en attente d'une erreur (je ne m'attarderai pas sur cette liste)
- *timeout* : le temps pendant lequelle la fonction attend avant de retourner. Si vous précisez en timeout *0*, la fonction retourne immédiatement. Si ce paramètre n'est pas précisé, la fonction retourne dès qu'un des sockets change d'état (est prêt à être lu si il est dans *rlist* par exemple), mais pas avant.

Concrètement, nous allons surtout nous intéresser au premier et au quatrième paramètre. En effet, *wlist* et *xlist* ne nous intéresseront pas présentement.

Nous, ce qu'on veut, c'est mettre des sockets dans une liste et que *select* les surveille, en retournant dès qu'un socket est prêt à être lu. Comme ça notre programme ne bloque pas et il peut recevoir des messages de plusieurs clients dans un ordre complètement inconnu .

Maintenant, concernant le *timeout*. Comme je vous l'ai dit, si vous ne le précisez pas, *select* bloque jusqu'au moment où un des sockets que l'on écoute est prêt à être lu, dans notre cas. Si vous précisez un *timeout* de *0*, *select* retournera tout de suite. Sinon, *select* retournera après le temps que vous indiquez en seconde, ou plus tôt si un socket est prêt à être lu.

En gros, si vous précisez un *timeout* de *1*, la fonction va bloquer pendant une seconde maximum. Mais si un des sockets en écoute est prêt à être lu dans l'intervalle (c'est-à-dire si un des clients envoie un message au serveur), la fonction retourne prématurément.

select retourne trois listes, là encore *rlist*, *wlist* et *xlist*. Sauf qu'au lieu d'être les listes que vous avez entrées, celles qui sont retournées contiennent uniquement les sockets "à lire" dans le cas de *rlist*.

Ce n'est pas clair ? Considérez cette ligne (ne l'essayez pas encore) :

Code : Python

```
rlist, wlist, xlist = select.select(clients_connectes, [], [], 2)
```

Cette instruction va écouter les sockets contenus dans la liste *clients_connectes*. Elle retournera au plus tard dans 2 secondes. Mais elle retournera plus tôt si un client envoie un message. La liste des clients ayant envoyé un message se retrouve dans notre variable *rlist*. On la parcourt ensuite et on peut appeler *recv* sur chacun des sockets.

Si ce n'est pas plus clair, rassurez-vous : nous allons voir *select* en action un peu plus bas. vous pouvez également aller jeter un coup d'oeil à la documentation du module.

select en action

Nous allons un peu travailler sur notre serveur. Vous pouvez garder le même client de test.

Le but va être de créer un serveur pouvant accepter plusieurs clients, réceptionner leurs messages et leur envoyer une confirmation à chaque réception. L'exercice ne change pas beaucoup, mais on va utiliser *select* pour travailler avec plusieurs clients.

J'ai parlé de *select* pour écouter plusieurs clients connectés, mais elle va également nous permettre de savoir si un (ou plusieurs) clients sont connectés au serveur. Si vous vous souvenez, la méthode *accept* est aussi une fonction bloquante. On va au reste l'utiliser de la même façon qu'un peut plus haut.

Je crois vous avoir donné assez d'informations théoriques. Le code doit parler maintenant :

Code : Python

```
import socket
import select

hote = ''
port = 12800

connexion_principale = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
connexion_principale.bind((hote, port))
connexion_principale.listen(5)
print("Le serveur écoute à présent sur le port {}".format(port))

serveur_lance = True
clients_connectes = []
while serveur_lance:
    # On va vérifier que de nouveaux clients ne demandent pas à se
connecter
    # Pour cela, on écoute notre connexion_principale en lecture
    # On attend maximum 50ms
    connexions_demandees, wlist, xlist =
select.select([connexion_principale],
        [], [], 0.05)

    for connexion in connexions_demandees:
        connexion_avec_client, infos_connexion = connexion.accept()
        # On ajoute le socket connecté à la liste des clients
        clients_connectes.append(connexion_avec_client)

    # Maintenant, on écoute la liste des clients connectés
    # Les clients retournés par select sont ceux devant être lu
(recv)
    # On attend là encore 50ms maximum
    # On enferme notre appel à select.select dans un bloc try
    # En effet, si notre liste de clients connectés est vide, une
exception
    # peut être levée
    clients_a_lire = []
    try:
        clients_a_lire, wlist, xlist =
select.select(clients_connectes,
                [], [], 0.05)
    except select.error:
        pass
    else:
        # On parcourt la liste des clients à lire
```

```
            for client in clients_a_lire:
                # client est de type socket
                msg_recu = client.recv(1024)
                # peut planter si le message contient des caractères
    spéciaux
                msg_recu = msg_recu.decode()
                print("Reçu {}".format(msg_recu))
                client.send(b"5 / 5")
                if msg_recu == "fin":
                    serveur_lance = False

    print("Fermeture des connexions")
    for client in clients_connectes:
        client.close()

    connexion_principale.close()
```

C'est plus long hein ? C'est inévitable, cependant.

Maintenant notre serveur peut accepter des connexions de plus d'un client, vous pouvez faire le test. En outre, il ne se bloque pas dans l'attente d'un message, du moins seulement 50 milisecondes maximum.

Je pense que les commentaires sont assez précis pour vous permettre d'aller plus loin. Ceci n'est naturellement pas encore une version complète, mais grâce à cette base vous devriez pouvoir facilement arriver à quelque chose. Pourquoi pas faire un mini-chat ?

Les déconnexions fortuites ne sont pas gérées non plus. Mais vous avez assez d'éléments pour faire des tests et améliorer notre serveur si cela vous tente .

Et encore plus

Je vous l'ai dit, le réseau est un vaste sujet, et même en se restreignant au sujet que j'ai choisi il y aurai beaucoup d'autres choses à vous montrer. Je ne peux tout simplement pas remplacer la documentation et donc, si vous voulez en apprendre plus, je vous invite à jeter un oeil à :

- La documentation du module socket
- La documentation du module select
- La documentation du module socketserver.

Le dernier module, *socketserver*, propose une alternative pour monter vos applications serveur. Il en existe, dans tous les cas, d'autres : vous pouvez utiliser des sockets non-bloquants (c'est-à-dire qui ne bloquent pas le programme quand vous utilisez leur méthode *accept* ou *recv*) ou des **threads** pour exécuter différentes portions de votre programme en parallèle. Mais je vous laisse vous documenter sur ces sujets si ils vous intéressent .

Voilà qui conclut notre approche des réseaux. Un sujet fascinant et dur à traiter en un seul chapitre, même pour un tour d'horizon assez bref. J'espère que cette approche vous aura en tous les cas permi de comprendre les principaux mécanismes de communication des sockets et vous aura donné une base d'information si vous souhaitez faire quelque chose de plus complet que les exemples proposés .

Et ici s'arrête cette partie sur la librairie standard. Vous vous en doutez, il reste des tonnes de choses à voir, mais pour le reste, il vous faudra aller jeter un oeil du côté de la documentation officielle de Python, en anglais, mais complète !

Encore une fois, je ne vous ai montré que quelques modules intéressants de la librairie standard. Vous pouvez retrouver une liste complète des modules de cette fameuse librairie standard, classés par thème, à l'adresse http://docs.python.org/py3k/library/index.html.

N'hésitez pas à y jeter un coup d'oeil, d'autant si vous cherchez quelque chose de précis .

Partie 5 : Récapitulatif et annexes

Cette partie constitue une forme d'annexe récapitulant de façon sommaire ce que vous avez vu dans les parties précédentes. Il s'agit d'une série de résumés, très condensés, qui pourraient s'avérer utile si vous avez oublié à quoi sert tel mot-clé ou comment faire telle chose.

Il ne s'agira que d'un résumé : s'il ne suffit pas, reportez-vous aux parties précédentes .

Je vous donnerai également quelques conseils, quelques réponses à la grande question comment continuer et quelques petites indications sur des fonctionnalités annexes qui sortent un peu du cadre de ce cours

Écrire nos programmes Python dans des fichiers

Ce petit chapitre vous explique comment mettre votre code Python dans un fichier pour l'exécuter. Vous pouvez lire ce chapitre très rapidement tant que vous savez à quoi sert la fonction *print*, c'est tout ce dont vous avez besoin.

Mettre le code dans un fichier

Pour mettre du code dans un fichier que nous pourrons ensuite exécuter, la démarche est très simple :

Ouvrez un éditeur standard sans mise en forme (Notepad2, Notepad++, VIM, Emacs…). Dans l'absolu, le bloc-notes Windows est aussi candidat, mais il reste moins agréable pour programmer (pas de coloration syntaxique du code, notamment).

Dans ce fichier, recopiez simplement la ligne :

Code : Python

```
print("Bonjour le monde !")
```

Enregistrez ce code dans un fichier à l'extension *.py*. Cela est surtout utile sous Windows.

Exécuter notre code sous Windows

Dans l'absolu, vous pouvez double-cliquer sur le fichier à l'extension *.py*, dans l'explorateur de fichiers. Mais la fenêtre s'ouvre et se referme très rapidement. Vous avez trois possibilités pour l'éviter :

- Mettre en pause le programme (voir la dernière sous-partie de ce chapitre)
- Lancer le programme depuis la console Windows (je ne m'y attarderai pas ici)
- Exécuter le programme avec **IDLE**.

C'est cette dernière opération que je vais détailler brièvement. Faites un clic droit sur le fichier *.py*. Vous devriez avoir quelque chose comme *edit with IDLE*. Cliquez dessus.

La fenêtre d'IDLE devrait alors s'afficher :

Vous pouvez voir votre code, ainsi que plusieurs boutons. Cliquez sur *run* puis sur *run module* (ou appuyez sur F5 directement).

Le code du programme devrait se lancer. Cette fois, la fenêtre console reste ouverte pour que vous puissiez voir le résultat ou les erreurs éventuelles.

Sous les systèmes Unix

Il est nécessaire d'ajouter une ligne tout en haut de votre programme indiquant le chemin menant vers l'interpréteur. Elle se présente sous la forme : *#!/adresse*.

Les habitués du BASH devraient reconnaître cette ligne assez rapidement. Pour les autres, sachez qu'il suffit de mettre à la place de « adresse » le chemin absolu de l'interpréteur (le chemin qui en partant de la racine, mène à l'interpréteur Python). Par

exemple :

Code : Python

```
#!/usr/bin/python3.2
```

En changeant les droits d'accès en exécution sur le fichier, vous devriez pouvoir le lancer directement.

Préciser l'encodage de travail

A partir du moment où vous mettez des accents dans votre programme, vous devrez préciser l'encodage que vous utilisez pour écrire votre programme.

Décrit très brièvement, l'encodage est une table contenant une série de codes symbolisant différents accents. Il existe deux encodages très utilisés : l'encodage *Latin-1* sous Windows et l'encodage *Utf-8* que l'on retrouve surtout sous les machines Unix.

Vous devez préciser à Python dans quel encodage vous écrivez votre programme. La plupart du temps, sous Windows, ce sera donc *Latin-1*, alors que sous Linux et Mac ce sera plus vraissemblablement *Utf-8*.

Une ligne de commentaire doit être ajoutée tout en haut de votre code (si vous êtes sous Unix, sous la ligne précisant le chemin menant vers l'interpréteur). Cette ligne s'écrit ainsi :

Code : Python

```
# -*-coding:encodage -*
```

Remplacez *encodage* par l'encodage que vous utilisez en fonction de votre système.

Sous Windows, on trouvera donc plus vraissemblablement : `# -*-coding:Latin-1 -*`

Sous Linux ou Mac, ce sera plus vraissemblablement : `# -*-coding:Utf-8 -*`

Gardez la ligne qui marche chez vous et n'oubliez pas de la mettre en tête de chacun de vos fichiers exécutables Python.

Pour en savoir plus sur l'encodage, je vous renvoie à l'adresse http://www.siteduzero.com/tutoriel-3-1 [...] -unicode.html . Vous y trouverez plein d'informations utiles .

Mettre en pause notre programme

Sous Windows se pose un problème : si vous lancez votre programme en double-cliquant dessus directement dans l'interpréteur, il aura tendance à s'ouvrir et se fermer très rapidement. Python exécute bel et bien le code, affiche le résultat, mais tout ça très rapidement. Et une fois que la dernière ligne de code a été exécutée, Windows ferme la console .

Pour pallier ce problème, on peut demander à Python de se mettre en pause à la fin de l'exécution du code.

Il va falloir rajouter deux lignes, l'une au début de notre programme et l'autre toute à la fin. La première importe le module *os* et la seconde utilise une fonction de ce module pour mettre en pause le programme. Si vous ne savez pas ce qu'est un module, ne vous en faites pas : le code suffira, un chapitre dans la première partie vous expliquera de ce dont il s'agit .

Code : Python

```
# -*-coding:Latin-1 -*
import os # on importe le module os
print("Bonjour le monde !")
os.system("pause")
```

Ce sont la ligne 2 et la ligne 4 qui sont nouvelles pour nous et que vous devez retenir. Quand vous exécutez ce code, vous obtenez :

Code : Console

```
Bonjour le monde !
Appuyez sur une touche pour continuer...
```

Vous pouvez donc lancer ce programme en double-cliquant directement dessus dans l'explorateur de fichier.

Ce code ne marche que sous Windows !

Si vous voulez mettre en pause votre programme sous Linux ou Mac, vous devrez utiliser un autre moyen. Ce n'est pas le mieux, car la fonction *input* n'est pas fait pour, mais vous pouvez conclure votre programme par la ligne :

Code : Python

```
input("Appuyez sur ENTREE pour fermer ce programme...")
```

Vous savez à présent comment mettre votre code Python dans un fichier et comment exécuter ce fichier. Si vous n'arrivez pas à bien exécuter le fichier, n'hésitez pas à demander de l'aide sur le forum.

Je vous conseille également de passer par IDLE plutôt que de double-cliquer directement sur le fichier *.py*. S'il existe des erreurs dans votre code, IDLE vous les indiquera, ce qui n'est pas négligeable.

De bonnes pratiques

Nous allons à présent nous intéresser à quelques **bonnes pratiques** quand on code en Python.

Les conventions que nous allons voir sont, naturellement, des propositions. Vous pouvez coder en Python sans les suivre .

Toutefois, prenez le temps de considérer les quelques affirmations ci-dessous. Si vous ne vous sentez concerné ne serai-ce que par une d'entres elles, je vous invite à lire ce chapitre .

- Un code dont on est l'auteur peut être difficile à relire si on l'abandonne quelques temps
- Lire le code d'un autre développeur est toujours plus délicat
- Si votre code doit être utilisé par d'autres, il doit être facile à reprendre (à lire et à comprendre).

Vous avez absolument le droit de répondre en disant que personne ne lira votre code de toute façon et que vous n'aurez aucun mal à comprendre votre propre code. Seulement, si votre code prend des proportions, si l'application que vous développez devient de plus en plus utilisée ou si vous vous lancez dans un gros projet, il est préférable pour vous d'adopter quelques conventions clairement définies dès le début. Et étant donné qu'il n'est jamais certain qu'un projet, démarré comme un amusement passager, ne devienne pas un jour énorme, ayez le réflexe dès le début .

En outre, vous ne pouvez jamais être sur à cent pourcent qu'aucun développeur ne vous rejoindra sur le projet à terme. Si votre application est utilisée, là encore, ce jour arrivera peut-être, où vous n'aurez pas assez de temps pour poursuivre, seul, son développement.

Quoiqu'il en soit, je vais vous présenter plusieurs conventions qui nous sont proposées au travers de PEP (Python Enhancement Proposal : propositions d'améliorations de Python). Encore une fois, il s'agit de **propositions** et vous pouvez choisir d'autres conventions si celles-ci ne vous plaisent pas .

La PEP 20 : toute une philosophie

La PEP 20, intitulée The Zen of Python, nous donne des conseils très généraux sur le développement.

Bien entendu, ce sont plus des conseils axés sur "comment programmer en Python" mais la plupart d'entre eux peuvent s'appliquer à la programmation en général.

Vous pouvez trouver le texte original (en anglais) de cette PEP ici : http://www.python.org/dev/peps/pep-0020/.

Je vous propose une traduction de cette PEP :

- Beautiful is better than ugly : beau est préférable à laid
- Explicit is better than implicit : explicite est préférable à implicite
- Simple is better than complex : simple est préférable à complexe
- Complex is better than complicated : complexe est préférable à compliqué
- Flat is better than nested : étalé[1] est préférable à imbriqué
- Sparse is better than dense : aéré est préférable à compact
- Readability counts : la lisibiilité compte
- Special cases aren't special enough to break the rules : les cas particuliers ne sont pas suffisamment particuliers pour casser la règle
- Although practicality beats purity : bien que la praticité heurte[2] la pureté
- Errors should never pass silently : les erreurs ne devraient jamais passer silencieusement
- Unless explicitly silenced : à moins qu'elles n'aient été explicitement mises au silence
- In the face of ambiguity, refuse the temptation to guess : en cas d'ambiguité, résistez à la tentation de deviner
- There should be one-- and preferably only one --obvious way to do it : il devrait exister une (et de préférence seulement une) façon de faire une chose
- Although that way may not be obvious at first unless you're Dutch : Même si cette façon pourrait ne pas être évidente de prime abord sauf si vous êtes néerlandais
- Now is better than never : **maintenant** est préférable à **jamais**
- Although never is often better than *right* now : mais **jamais** est parfois préférable à **immédiatement**
- If the implementation is hard to explain, it's a bad idea : si l'implémentation est difficile à expliquer, c'est une mauvaise idée
- If the implementation is easy to explain, it may be a good idea : si l'implémentation est facile à expliquer, ce peut être une bonne idée
- Namespaces are one honking great idea -- let's do more of those : les espaces de noms sont une très bonne idée (faisons-en plus !)<

- [1] Moins littéralement, du code trop imbriqué (une fonction appelant une fonction appelant une fonction... par exemple) est plus difficile à lire
- [2] Moins littéralement, il est difficile de faire un code pratique (fonctionnel) et "pur".

Comme vous le voyez, c'est une liste d'aphorismes très simples. Ils donnent des idées sur le développement Python, mais en la lisant pour la première fois vous n'y voyez sans doute que peu de conseils pratiques.

Cependant, cette liste d'aphorismes est vraiment importante et peut être très utile. Certaines des idées qui s'y trouvent comportent des pans entiers de la philosophie de Python.

Si vous travaillez sur un projet en équipe, un autre développeur pourra contester une implémentation quelconque en se fiant sur un des aphorismes cités plus haut.

Quand bien même vous travaillerez seul, la philosophie de Python est ici et c'est toujours mieux de comprendre et appliquer la philosophie d'un langage quand on l'utilise pour du développement.

Je vous conseille donc de la garder sous les yeux autant que possible et de vous y référer à la moindre occasion .
Commencez par lire chaque proposition. Les lignes sont courtes, prenez le temps de bien comprendre ce qu'elles veulent dire.

Sans trop détailler ce qui se trouve au-dessus (cela prendrai trop de temps), je signale à votre attention que plusieurs des aphorismes que nous venons de voir parlent surtout de l'allure du code. L'idée qui semble se dissimuler derrière, c'est qu'un code fonctionnel n'est pas suffisant : il faut, autant que possible, faire du "beau code". Qui marche, naturellement... mais qu'il marche n'est pas suffisant .

Maintenant, nous allons nous intéresser à deux autres PEP qui vous donnent des conseils très pratiques sur votre développement.

- La première nous donne des conseils très précis sur la présentation du code
- La seconde nous donne des conseils sur la documentation au coeur de notre code.

La PEP 8 : des conventions précises

Maintenant que nous avons vu des directives très générales, nous allons nous intéresser à une autre proposition d'amélioration, la PEP 8. Elle nous donne des conseils très précis sur la forme du code. Là encore, c'est à vous de voir : vous pouvez appliquer totalement les conseils donnés ici, ou seulement partiellement .

Je ne vais pas reprendre tout ce qui est dit dans cette PEP mais je vais reprendre la plupart des conseils en les simplifiant. Si donc une des propositions que vous pouvez voir en lisant cette sous-partie manque d'explications à vos yeux, je vous conseille d'aller faire un tour sur la PEP originale : http://www.python.org/dev/peps/pep-0008/. Ce qui suit n'est pas une traduction complète, j'insiste sur ce point .

Introduction

L'une des convictions de Guido [1] est que le code est lu beaucoup plus souvent qu'il n'est écrit. Les conseils donnés ici sont censés améliorer la lisibilité du code. Comme le dit la PEP 20, **la lisibilité compte**.

- [1] Guido Van Rossum, créateur et BDFL (Benevolent Dictator For Life : dictateur bienveillant à vie) de Python.

Un guide comme celui-ci parle de cohérence. La cohérence au coeur d'un projet est importante. La cohérence au sein d'une fonction ou d'un module est encore plus importante.

Mais encore plus important, sachez quand être incohérent (parfois, les conseils de style donnés ici ne s'appliquent pas). En cas de doute, remettez-vous en à votre bon sens. Regardez plusieurs exemples et choisissez lequel semble le meilleur.

Deux bonnes raisons pour ne pas respecter une règle donnée :

1. Quand appliquer la règle rendrai le code moins lisible
2. Dans un soucis de cohérence avec du code existant qui ne respecte pas cette règle non plus. Ce cas peut se produire si vous utilisez un module ou une bibliothèque qui ne respecte pas les mêmes conventions que celles définies ici.

Forme du code

- Indentation : utilisez 4 espaces par niveau d'indentation
- Tabulations ou espaces : ne mélangez jamais dans le même projet des indentations avec des espaces et des tabulations. A choisir, les espaces sont généralement préférées mais les tabulations peuvent être également utilisées pour marquer l'indentation
- Longueur maximum d'une ligne : limitez vos lignes à un maximum de 79 caractères. De nombreux éditeurs utilisent préférenciellement des lignes de 79 caractères maximum. Pour de longs blocs de texte (docstrings par exemple), limitez-vous de préférence à 72 caractères par ligne.

  Quand cela est possible, découpez vos lignes en utilisant des parenthèses, crochets ou accolades plutôt que l'antislash \.

  Exemple :

  Code : Python

```
appel_d_une_fonction(parametre_1, parametre_2,
        parametre_3, parametre_4):
    ...
```

  Si vous devez découper une ligne trop longue, faites la césure après l'opérateur, pas avant.

  Code : Python

```
# oui
    un_long_calcul = variable + \
            taux * 100

# non
    un_long_calcul = variable \
            + taux * 100
```

- Sauts de ligne : séparez de deux sauts de ligne la définition d'une fonction et la définition d'une classe.

  Les définitions de méthode au coeur d'une classe sont séparées par une ligne vide.

  Des sauts de ligne peuvent également être utilisés, parcimonieusement, pour délimiter des portions de code
- Encodages : à partir de Python 3.0, il est conseillé d'utiliser, dans du code comportant des accents, l'encodage Utf-8.

Directives d'importation

- Les directives d'importation doivent préférenciellement se trouver sur plusieurs lignes. Par exemple :

  Code : Python

```
import os
import sys
```

Plutôt que :

Code : Python

```
import os, sys
```

Cette syntaxe est cependant acceptée quand on importe certaines données d'un module :

Code : Python

```
from subprocess import Popen, PIPE
```

- Les directives d'importation doivent toujours se trouver en tête du fichier, sous la documentation éventuelle du module mais avant la définition de globales ou constantes du module
- Les directives d'importation doivent être divisées en trois groupes, dans l'ordre :

    1. Les directives d'importation faisant référence à la librairie standard
    2. Les directives d'importation faisant référence à des bibliothèques tierces
    3. Les directives d'importation faisant référence à des modules de votre projet.

    Une ligne devrait être sautée entre chaque groupe de directives d'importation
- Préférez utiliser des chemins absolus plutôt que relatifs dans vos directives d'importation. Autrement dit :

Code : Python

```
from paquet.souspaquet import module

# est préférable à
from . import module
```

Le signe espace dans les expressions et instructions

- Evitez le signe espace dans les situations suivantes :
    - Au coeur des parenthtèses, crochets et accolades :

Code : Python

```
# oui
    spam(ham[1], {eggs: 2})

# non
    spam( ham[ 1 ], { eggs: 2 } )
```

    - Juste avant une virgule, un point virgule ou un signe deux points :

Code : Python

```
# oui
    if x == 4: print x, y; x, y = y, x

# non
    if x == 4 : print x , y ; x , y = y , x
```

    - Juste avant la parenthèse ouvrante listant les paramètres d'une fonction :

Code : Python

```
# oui
    spam(1)

# non
    spam (1)
```

- Juste avant le crochet ouvrant indiquant une indexation ou sélection :

Code : Python

```
# oui
    dict['key'] = list[index]

# non
    dict ['key'] = list [index]
```

- Plus d'un espace autour de l'opérateur d'affectation = (ou autre) pour l'aligner avec une autre instruction :

Code : Python

```
# oui
    x = 1
    y = 2
    long_variable = 3

# non
    x             = 1
    y             = 2
    long_variable = 3
```

- Toujours entourer les opérateurs suivants d'un espace (un avant le symbole, un après) :
  - Affectation : =, +=, -=, ...
  - Comparaison : <, >, <=, ..., *in*, *not in*, *is*, *is not*
  - Booléens : *and*, *or*, *not*
  - Arithmétiques : +, -, *, ...

Code : Python

```
# oui
    i = i + 1
    submitted += 1
    x = x * 2 - 1
    hypot2 = x * x + y * y
    c = (a + b) * (a - b)

# non
    i=i+1
    submitted +=1
    x = x*2 - 1
    hypot2 = x*x + y*y
    c = (a+b) * (a-b)
```

Attention : n'utilisez pas d'espaces autour du signe = si c'est dans un contexte de paramètre à valeur par défaut (définition d'une fonction) ou d'appel de paramètre (appel de fonction).

Code : Python

```
# oui
    def fonction(parametre=5):
        ...
    fonction(parametre=32)

# non
    def fonction(parametre = 5):
        ...
    fonction(parametre = 32)
```

- Plusieurs instructions sur une même ligne sont déconseillées :

Code : Python

```
# oui
    if foo == 'blah':
        do_blah_thing()
    do_one()
    do_two()
    do_three()

# plutôt que
    if foo == 'blah': do_blah_thing()
    do_one(); do_two(); do_three()
```

Commentaires

-  Les commentaires qui contredisent le code sont pires qu'aucun commentaire. Faites toujours une de vos priorités la mise à jour des commentaires quand le code change !

- Les commentaires devraient être des phrases complètes, commençant par une majuscule. Le point terminant la phrase peut être absent si le commentaire est court
- Si vous écrivez en anglais, Strunk and White s'applique
- A l'intention des codeurs non-anglophones : s'il vous plaît, écrivez vos commentaires en anglais, sauf si vous êtes sûr à 120%  que votre code ne sera pas lu par quelqu'un qui ne comprend pas votre langue.

Conventions de nommage

Noms à éviter

N'utilisez jamais les caractères suivants, seuls comme noms de variable : *l* (l minuscule), *O* (o majuscule) et *I* (i majuscule). Avec certaines polices, ces caractères peuvent être aisément confondus avec les chiffres *0* ou *1*.

Nom des modules et packages

Les modules et packages doivent avoir des noms courts constitués de lettres minuscules. Les noms de module peuvent contenir des signes _ (souligné). Bien que les noms de package puissent en contenir également, la PEP 8 nous le déconseille.

Noms de classes

Sans presque aucune exception, les noms de classes utilisent la convention suivante : la variable est écrite en minuscule, exceptée la première lettre de chaque mot la constituant. Par exemple : *MaClasse*.

Noms d'exceptions

Les exceptions étant des classes, elles suivent la même convention. En anglais, si l'exception est une erreur, on fait suivre le nom

du suffixe *Error* (vous retrouvez cette convention dans *SyntaxError*, *IndexError*...).

Noms de variables, fonctions et méthodes

La même convention est utilisée pour des noms de variable (instances d'objet), de fonctions ou de méthodes : le nom tout en minuscule et les mots séparés par des signes soulignés (_). Exemple : *nom_de_fonction*.

Constantes

Les constantes doivent être écrites tout en majuscule avec chaque mot séparé par un signe souligné (_). Exemple : *NOM_DE_MA_CONSTANTE*.

Conventions de programmation

Comparaisons

Les comparaisons avec des singletons (comme *None*) doivent toujours se faire avec les opérateurs *is* et *is not*, jamais avec les opérateurs == ou *!=*.

Code : Python

```
# oui
    if objet is None:
        ...

# non
    if objet == None:
        ...
```

Quand cela est possible, utilisez l'instruction `if objet:` quand vous voulez dire `if objet is not None:`.

La comparaison de type d'objets doit se faire avec la fonction *isinstance* :

Code : Python

```
# oui
    if isinstance(variable, str):
        ...


# non
    if type(variable) == str:
        ...
```

Quand vous comparez des séquences, utilisez le fait qu'une séquence vide est *False*.

Code : Python

```
if liste: # la liste n'est pas vide
```

Enfin, ne comparez pas des booléens à *True* ou *False* :

Code : Python

```
# oui
    if booleen: # si booleen est vrai
```

```
            ...
        if not booleen: # si booleen n'est pas vrai
            ...

    # non
        if booleen == True:
            ...

    # encore pire
        if booleen is True:
            ...
```

Conclusion

Voilà pour la PEP 8. Elle liste beaucoup de conventions et toutes ne sont pas présentes dans cette sous-partie. Celles qui le sont, dans tous les cas, sont moins détaillées. Je vous invite donc à faire un tour du côté du texte original : http://www.python.org/dev/peps/pep-0008/.

La PEP 257 : de belles documentations

Nous allons nous intéresser à présent à la PEP 257 qui définit d'autres conventions concernant la documentation via les docstrings.

Code : Python

```
def fonction(parametre1, parametre2):
    """Documentation de la fonction."""
```

La ligne 2 de ce code, comme vous l'avez sans doute reconnue, est une docstring. Nous allons voir quelques conventions autour de l'écriture de telles docstrings (comment les écrire, que préciser dedans...).

Une fois de plus, je vais prendre quelques libertés avec le texte original de la PEP. Je ne vous proposerai pas une traduction complète de la PEP mais je reviendrai sur les points que je considère comme importants. Si vous voulez lire le texte original, rendez-vous à l'adresse http://www.python.org/dev/peps/pep-0257/.

Qu'est-ce qu'une docstring ?

La docstring (chaîne de documentation, en français) est une chaîne de caractères placée juste après la définition d'un module, d'une classe, fonction ou méthode. Cette chaîne de caractères devient l'attribut spécial *__doc__* de l'objet.

Code : Python Console

```
>>> fonction.__doc__
'Documentation de la fonction.'
>>>
```

Tous les modules doivent être documentés grâce aux docstrings. Les fonctions et classes exportées par un module doivent également être documentées ainsi. Cela est aussi vrai pour les méthodes publiques d'une classe (y compris le constructeur *__init__*). Un package peut être documenté via une docstring placée dans le fichier *__init__.py*.

Pour des raisons de cohérence, utilisez toujours des triple-quotes *"""* autour de vos docstrings. Utilisez *"""chaîne de documentation"""* si votre chaîne comporte des antislash \.

On peut trouver deux formes de docstring :

- Les docstrings sur une seule ligne
- Les docstrings sur plusieurs lignes.

Les docstrings sur une seule ligne

Code : Python

```
def kos_root():
    """Return the pathname of the KOS root directory."""
    global _kos_root
    if _kos_root: return _kos_root
    ...
```

Notes

1. Les triple-quotes sont utilisées même si la chaîne tient sur une seule ligne. Il est plus simple de l'étendre par la suite dans ces conditions
2. Les trois guillemets """ fermant la chaîne sont sur la même ligne que les trois guillemets l'ouvrant. Ceci est préférable pour une docstring d'une seule ligne
3. Il n'y a aucun saut de ligne avant ou après la docstring
4. La chaîne de documentation est une phrase, elle se termine par un point .
5. La docstring sur une seule ligne ne doit pas décrire la signature des paramètres à passer à la fonction / méthode, ou son type de retour. Ne faites pas :

Code : Python

```
def fonction(a, b):
    """fonction(a, b) -> list"""
```

Cette syntaxe est uniquement valable pour les fonctions C (comme les built-ins). Pour les fonctions Python, l'introspection peut être utilisée pour déterminer les paramètres attendus.

L'introspection ne peut cependant pas être utilisée pour déterminer le type de retour de la fonction / méthode. Si vous voulez le préciser, incluez-le dans la docstring sous une forme explicite :

Code : Python

```
"""Fonction faisant cela et retournant une liste."""
```

Bien entendu, "faisant cela" doit être remplacée par une description utile de ce que fait la fonction !

Les docstrings sur plusieurs lignes

Les docstrings sur plusieurs lignes sont constituées d'une première ligne résumant brièvement l'objet (fonction, méthode, classe, module), suivie d'un saut de ligne, suivi d'une description plus longue. Respectez autant que faire ce peut cette convention : une ligne de description brève, un saut de ligne puis une description plus longue.

La première ligne de la docstring peut se trouver juste après les guillemets ouvrant la chaîne ou juste en-dessous.
Dans tous les cas, le reste de la docstring doit être indenté au même niveau que la première ligne :

Code : Python

```
class MaClasse:
    def __init__(self, ...):
        """Constructeur de la classe MaClasse

Une description plus longue...
sur plusieurs lignes...

"""
```

Insérez un saut de ligne avant et après chaque docstring documentant une classe.

La docstring d'un module doit généralement lister les classes, exceptions et fonctions, ainsi que les autres objets exportés par ce module (une ligne de description par objet). Cette ligne de description donne généralement moins d'informations sur l'objet que sa propre documentation. La documentation d'un package (la docstring se trouvant dans le fichier *__init__.py*) doit également lister les modules et sous-packages exportés par lui.

La documentation d'une fonction ou méthode doit décrire son comportement et documenter ses arguments, sa valeur de retour, ses effets de bord, ses exceptions levées et les restrictions concernant son appel (quand ou dans quelle condition appeler cette fonction). Les paramètres optionnels doivent également être documentés.

Code : Python

```
def complexe(reel=0.0, image=0.0):
    """Forme un nombre complexe.

Paramètres nommés :
reel -- la partie réelle (0.0 par défaut)
image -- la partie imaginaire (0.0 par défaut)

"""
    if image == 0.0 and reel == 0.0: return complexe_zero
    ...
```

La documentation d'une classe doit, de même, décrire son comportement, documenter ses méthodes publiques et attributs.

Le BDFL nous conseille de sauter une ligne avant de fermer nos docstrings quand elles sont sur plusieurs lignes. Les trois guillemets fermant la docstring sont ainsi sur une ligne vide par ailleurs.

Code : Python

```
def fonction():
    """Documentation brève sur une ligne.

Documentation plus longue...

"""
```

Je ne saurai que trop vous conseiller d'aller consulter les textes originaux des trois PEP que nous avons vues :

- La PEP 20 : http://www.python.org/dev/peps/pep-0020/
- La PEP 8 : http://www.python.org/dev/peps/pep-0008/
- La PEP 257 : http://www.python.org/dev/peps/pep-0257/.

C'est peut-être en anglais, mais ce reste des textes originaux. Pour des raisons de concisions, j'ai fais plusieurs approximations et ommissions volontaires dans ce chapitre et je vous conseille vivement, à un moment ou à un autre, d'aller lire ces trois PEP en anglais  . Cela étant dit, pour ceux qui ne veulent pas se mouiller les pieds, ils auront toujours ce chapitre comme base pour quelques bonnes pratiques  .

Dans le prochain chapitre, le dernier de ce tutoriel, je vous présenterai quelques idées pour concrétiser vos projets informatiques via Python  . Nous y verrons notamment des bibliothèques pour faire des jeux ou des interfaces graphiques  .

Pour finir et bien continuer

La fin de ce cours sur Python approche. Mais si ce langage vous a plu, vous aimeriez probablement concrétiser vos futurs projets avec lui. Pour ce faire, je vous donne ici quelques indications.

Ce sera cependant en grande partie à vous d'explorer les pistes que je vous propose. Vous avez à présent bien assez de bagage pour vous lancer à corps perdu dans un projet d'une certaine importance, tant que vous en sentez la motivation.

Nous allons commencer par voir quelques-unes des ressources que l'on peut trouver sur Python, pour compléter vos connaissances sur ce langage.

Nous verrons ensuite plusieurs bibliothèques tierces spécialisées dans certains domaines, nous permettant de réaliser des interfaces graphiques par exemple.

Quelques références

Dans cette sous-partie, je vais surtout parler des ressources officielles que l'on peut trouver sur le site de Python.

Il en existe bien entendu d'autres, certaines d'entre elles sont en français. Mais les ressources les plus à jour concernant Python se trouvent sur le site de Python lui-même.

En outre, les ressources mises à disposition sont clairement expliquées, détaillées avec assez d'exemples pour comprendre leur utilité. Elles n'ont qu'un inconvénient : elles sont en anglais. Mais c'est le cas d'une majorité des documentations en programmation et il faudra bien envisager, un jour où l'autre, de s'y mettre pour aller plus loin .

La documentation officielle

Nous avons déjà parlé de la documentation officielle dans ces pages. Nous allons maintenant voir comment elle se décompose exactement.

Commencez par vous rendre sur le site de Python à l'adresse www.python.org.

Dans le menu de navigation, vous pourrez trouver plusieurs liens (notamment le lien de téléchargement, **DOWNLOAD**, sur lequel vous avez probablement cliqué pour télécharger Python). Il s'y trouve également le lien **DOCUMENTATION** et c'est sur celui-ci que je vous invite à cliquer à présent .

Dans la nouvelle page qui s'affiche, vous pouvez voir deux nouvelles informations :

- Sous le lien **DOCUMENTATION** dans le menu, un sous-menu apparaît à présent, contenant les liens **Current Docs**, **License**, **Help**...
- La page principale contient à présent les informations sur les documentations par version de Python. Par défaut, les deux versions de Python les plus récentes (dans la branche 2.X et 3.X) sont affichées mais vous pouvez afficher toutes les versions en cliquant sur le lien **the complete list of documentation by Python version** .

Nous allons nous intéresser d'abord au sous-menu.

Le Wiki Python

Python propose un Wiki en ligne qui centralise de nombreuses informations autour de Python. Si vous vous rendez sur le wiki, après avoir cliqué sur le lien **Wiki** dans le sous-menu, vous vous retrouvez sur une nouvelle page.

Il n'existe pas de version traduite du Wiki, vous devez donc demander à accéder à la page d'accueil en anglais.

Une fois là, vous pouvez trouver plusieurs informations classées par catégorie. Je vous laisse explorer si vous êtes intéressés .

L'index des PEP (Python Enhancement Proposal)

Dans ce sous-menu, vous pouvez également trouver un lien **PEP Index**. Si vous cliquez dessus, vous obtenez un tableau, ou plutôt plusieurs tableaux reprenant les PEPs classées par catégorie. Il y en a un paquet, comme vous pouvez le constater et

dans ce tutoriel, je n'ai pu vous parler que de quelques-unes . Libre à vous de parcourir cet index et de regarder certaines des PEP en fonction des sujets qui vous intéressent plus particulièrement.

La documentation par version

A présent, revenez dans la page de documentation de Python (www.python.org -> **DOCUMENTATION**).

Choisissez votre version actuelle de Python (**Browse Python 3.2.1 Documentation** pour moi).

Sur la nouvelle page qui s'affiche sous vos yeux, vous trouvez les grandes catégories de la documentation. En voici quelques-unes :

- **Tutorial** : Le tutoriel. Selon toute probabilité, les bases de Python vous sont acquises, cependant il est toujours utile d'aller faire un tour sur cette page pour regarder la table des matières
- **Library Reference** : la référence de la bibliothèque standard, nous reviendrons plus bas à cette page. Le conseil donné me paraît bon à suivre :

> **Citation**
>
> Keep this under your pillow

C'est-à-dire, "gardez-la sous votre oreiller"

- **Language Reference** : cette page décrit d'une façon très explicite la syntaxe du langage
- **Python HOWTOs** : une page regroupant plusieurs aides traitant de sujet très précis, comme comment bien utiliser les sockets.
- ...

Vous pouvez aussi trouver un classement par index que je vous laisse découvrir. Vous pourrez y voir, notamment, le lien permettant d'afficher la table des matières complète de la documentation . Vous y trouverez également un glossaire, utile dans certains cas .

La référence de la bibliothèque standard

Vous vous êtes peut-être déjà rendu sur cette page. Je l'espère en vérité . Elle comporte la documentation des types prédéfinis par Python, des fonctions **builtins** et exceptions, mais aussi des modules que l'on peut trouver dans la bibliothèque standard de Python. Ces modules sont classés par catégorie et il est assez facile (et parfois très utile) de survoler la table des matières pour savoir ce que Python nous permet de faire, sans installer de bibliothèque tierce.

C'est déjà pas mal, comme vous pouvez le voir !

Cela dit, il existent certains cas où des bibliothèques tierces sont nécessaires. Nous allons voir quelques-uns de ces cas dans la suite de ce chapitre, ainsi que quelques bibliothèques utiles dans ces circonstances.

Des bibliothèques tierces

La bibliothèque standard de Python comporte déjà beaucoup de modules et beaucoup de fonctionnalités. Mais il arrive, pour certains projets, que la bibliothèque standard ne suffise pas.

Si vous avez besoin de faire une application avec une interface graphique, la bibliothèque standard vous propose un module appelé **tkinter**. Il existe toutefois d'autres moyens pour faire des interfaces graphiques, en faisant appel à des bibliothèques tierces.

Ces bibliothèques se présentent comme des packages ou modules que vous installez pour les rendre accessibles depuis votre interpréteur Python.

A l'heure où j'écris ces lignes, toutes les bibliothèques dont je vais parler ne sont pas nécessairement compatibles Python 3.X.

Les développeurs des dites bibliothèques ont généralement comme projet à plus ou moins long terme de passer leur code en Python 3.X. Si certains ont déjà franchi le pas, d'autres attendent encore et ce travail est plus ou moins long en fonction des dépendances de la bibliothèque.

Bref, tout cela change très vite et si je vous dit que telle bibliothèque n'est pas compatible avec Python3, il faudra entendre **pour l'instant**. A l'heure où vous lisez ces lignes, il est bien possible qu'une version compatible soit parue. Le changement se fait, lentement mais sûrement  .

Ceci étant posé, examinons quelques bibliothèques tierces. Il en existe un nombre incalculable et je ne parle naturellement que de certaines d'entre elles ici.

Pour faire une interface graphique

Nous avons parlé de **tkinter**. Il s'agit d'un module disponible par défaut dans la bibliothèque standard de Python. Elle se base sur la librairie **Tk** et propose de développer des interfaces graphiques.

Il est cependant possible qu'elle ne corresponde pas à vos besoins. Il existe plusieurs bibliothèques tierces qui permettent de développer des interfaces graphiques, parfois en proposant quelques bonus. En voici trois parmi d'autres :

- PyQt : une bibliothèque permettant le développement d'interfaces graphiques, actuellement en version 4. En outre, elle propose plusieurs packages gérant le réseau, le SQL (bases de données), un kit de développement Web... et bien d'autres choses. Soyez vigilant cependant : PyQt est distribuée sous plusieurs licences, commerciales ou non. Vous devrez tenir compte de ce fait si vous commencez à utiliser cette bibliothèque
- PyGTK : comme son nom l'indique, une bibliothèque faisant le lien entre Python et la bibliothèque **GTK** / **GTK+**. Elle est distribuée sous licence LGPL
- wxPython : une bibliothèque faisant le lien entre Python et la bibliothèque WxWidget

Ces informations ne vous permettent pas de faire un choix immédiat entre telle ou telle bibliothèque, j'en ai conscience. Aussi je vous invite à aller jeter un oeil du côté des sites des différents projets.

Ces trois bibliothèques ont l'avantage d'être multi-plateforme et, généralement, assez simples à apprendre. En fonction de vos besoins, vous vous tournerez plutôt vers l'une ou l'autre, mais je ne peux certainement pas vous aider dans ce choix  . Je vous invite donc à rechercher par vous-même si vous êtes intéressé.

Dans le monde du Web

Il existe là encore de nombreuses bibliothèques, bien que je ne parle que de deux ici.

Django

La première, dont vous avez sans doute entendu parlé auparavant, est Django.

Django est une bibliothèque, ou plutôt un Framework, permettant de développer votre site dynamiquement en Python. Il propose de nombreuses fonctionnalités que vous trouverez, je pense, aussi puissantes que flexibles si vous prenez le temps de vous pencher sur le site du projet.

A l'heure où j'écris ces lignes, certains développeurs tentent de proposer des patch pour adapter Django à Python 3. L'équipe du projet, cependant, ne prévoit pas dans l'immédiat de développer de branche pour Python 3.

Si vous tenez réellement à Django et qu'aucune version stable n'existe encore sous Python 3 à l'heure où vous lisez ces lignes, je vous encourage à tester cette bibliothèque sous Python 2.X. Rassurez-vous, vous n'aurez pas toute la syntaxe à apprendre pour programmer dans ce langage : les changements sont très clairement listés sur le site de Python et ne présentent pas un obstacle insurmontable pour qui est motivé  .

CherryPy

CherryPy est une bibliothèque permettant de construire tout votre site en Python. Elle n'a pas la même vocation que Django puisqu'elle permet de créer la hiérarchie de votre site dynamiquement mais vous laisse libre des outils à employer pour faire quelque chose de plus complexe.

Un peu de réseau

Pour finir, nous allons parler d'une bibliothèque assez connue, appelée Twisted.

Cette bibliothèque est orientée vers le réseau. Elle supporte de nombreux protocoles de communication réseau (le TCP et l'UDP, bien entendu, mais aussi l'HTTP, le SSH et de nombreux autres). Si vous voulez créer une application utilisant l'un de ces protocoles, Twisted pourrait être une bonne solution.

Là encore, je vous invite à jeter un oeil sur le site du projet pour plus d'informations. A l'heure où j'écris ces lignes, Twisted n'est utilisable que sous la branche 2.X de Python. Cependant, on peut trouver une future version en cours d'élaboration portant Twisted sur la branche 3.X .

Pour conclure

Ce ne sont là que quelques bibliothèques tierces, il en existe de nombreuses autres, certaines dédiées à des projets très précis. Je vous invite à faire des recherches plus avancées si vous avez des besoins plus spécifiques. Vous pouvez commencer avec la liste de bibliothèques qui se trouve au-dessus, avec les réserves suivantes :

- Je ne donne que peu d'informations sur chaque bibliothèque et elles ne s'accordent peut-être plus avec celles disponibles sur le site du projet. En outre, la documentation de chaque bibliothèque reste et restera une source plus sûre et actuelle, dans tous les cas
- Ces projets évoluent rapidement. Il est fort possible que ce que je dise des bibliothèques ne soit plus vrai à l'heure où vous lisez ces lignes. Pour mettre à jour ces informations, il n'y a qu'une solution imparable : allez sur le site des projets .

Une dernière petite parenthèse avant de nous quitter : j'ai essayé de présenter, tout au long de ce cours, des données utiles et à jour sur le langage de programmation Python, dans sa branche 3.X. Vous ne pouvez cependant pas, sur la base de ce tutoriel, estimer avoir tout vu de Python, même en excluant la bibliothèque standard et les bibliothèques tierces : le langage lui-même vous réserve encore des surprises, même si ce cours s'arrête ici .

C'est bien la fin. Je vous souhaite, dans tous les cas, une bonne continuation dans vos projets futurs avec Python . Ce cours s'arrête ici... mais l'histoire ne s'arrête pas là .

Cette fois, c'est la fin des fins. Je vous souhaite un bon développement et une bonne exploration des fonctionnalités, modules, librairies qu'il vous reste à découvrir. Pas de quartier !

Ce tutoriel est à présent achevé. J'espère sincèrement qu'il vous aura permis de vous familiariser avec la syntaxe de Python et de vous présenter quelques-uns des puissants outils associés à ce langage.

Le monde de la programmation s'ouvre à vous désormais et j'espère qu'il vous semblera aussi riche de potentiel qu'à moi .