Livre Python

Version 1.0-draft

Harold Erbin

18 February 2012

Table des matières

Version : 18 February 2012

Les mises à jour de ce livre et les corrections des exercices peuvent être trouvées sur le site : http ://harold.e.free.fr/python/.

N'hésitez pas à me contacter (harold.erbin@gmail.com) pour tout commentaire ou à créer un ticket sur Bitbucket (https ://bitbucket.org/Melsophos/tutoriel-python-3/issues/new) pour proposer une amélioration ou rapporter une erreur.

Première partie

Préface

CHAPITRE 1

Avant-propos

Ce livre a pour but d'offrir une large description du langage Python. Il abordera le langage lui-même, l'utilisation des interfaces graphiques (GTK et Qt), la création d'un site web (avec Django) ainsi que des applications aux sciences.

Ce projet est né suite à la constatation qu'il n'existait que peu de documentations françaises et récentes sur Python (même si celles qui existent sont d'une excellente qualité), et *a fortiori* sur les domainces avancés précédemment cités. Les parties sur ces derniers points seront rédigées avec le temps.

Le point de vue adopté sera résolument moderne, à savoir que seule la version 3 de Python sera traitée.

Enfin, il n'est pas nécessaire de connaitre un autre langage avant de pouvoir commencer à lire ce livre, car j'ai pris le parti de débuter par une introduction à la programmation en générale, avant de poursuivre en expliquant chaque détail de Python. Toutefois j'espère que même les connaisseurs y trouveront des informations précieuses.

Remerciements

– 19. Goldszmidt (remarques sur le contenu).

CHAPITRE 3

Plan de l'ouvrage

Introduction Cette partie vise à donner quelques informations sur la programmation en général, sans toutefois entrer dans les détails puisque d'autres ouvrages plus spécialisés existent (et l'information est abondante).

Le langage Cette partie abordera le langage lui-même et ses principaux concepts.

Aspects spécifiques Cette partie abordera des points spécifiques du langage, qui ne servent que dans certains cas particuliers.

GTK Cette partie abordera la création d'une interface graphique avec pyGTK.

Qt Cette partie abordera la création d'une interface graphique avec pyQt.

Django Cette partie concernera la création d'un site internet avec le framework Django.

Sciences Cette partie explicitera l'utilisation de Python dans un cadre scientifique. Il abordera aussi l'algorithmique.

Outils Dans cette partie seront détaillés tous les outils pouvant servir à l'utilisation de Python.

Annexes Enfin, cette dernière partie apportera des informations complémentaires.

Des exercices seront présents à la fin de chaque chapitres, afin de donner au lecteur quelques pistes pour s'entrainer, mais en aucun cas il ne doit se limiter aux exercices indiqués : la curiosité et les tests personnels sont très importants pour progresser. De plus, il ne faut pas s'empresser d'aller regarder la correction dès que l'on bloque. Au contraire, il faut persévérer, en passant éventuellement à autre chose avant d'y revenir : il est impossible de tout trouver en quelques secondes. De plus, je vous conseille de chercher s'il existe d'autres manières de concevoir les programmes des exercices : parfois il existe des manières plus simples ou plus rapides de programmer, que nous n'avions pas vu au début (la question de la rapidité sera abordée en détails dans la partie scientifique).

CHAPITRE 4

Conventions

4.1 Présentation du code

Les exemples de code seront affichés comme ceci `print a` en ligne, et sinon comme ceci :

```
print a
```

Parfois la code comportera les numéros de ligne ou sera mis en valeur (cadre par exemple).

Le symbole >>> (prompt primaire) indique une invite de commande en mode interactif (voir section *Interpréteurs interactifs* pour davantage d'explications), et ... (prompt secondaire) lorsqu'une commande s'étend sur plusieurs lignes. Enfin, dans le cas d'une invite de commande, l'affichage attendu est affiché juste en dessous.

```
>>> a = 1
>>> if a == 1:
...     print(a)
1
```

Il s'agira donc exactement de ce que le lecteur doit s'attendre à écrire et à voir lorsqu'il utilise Python dans le terminal classique.

Dans le cas d'un code destiné à être placé dans un fichier, le code ne sera pas précédé de l'invite de commande (et la sortie ne sera logiquement pas affichée dans le code).

```
def function():
    return a
function()
```

Enfin, lorsque la représentation d'un objet sera affichée, des points de suspension remplaceront son adresse (tout ceci sera vu en détails) : par exemple, au lieu de `<function func at 0xb7a9242c>`, il sera écrit `<function func at ...>`, car l'adresse change à chaque exécution.

Parfois il sera nécessaire de se servir du terminal (ou ligne de commandes ou encore shell — les trois termes seront utilisés) pour lancer une commande bash. En ce cas, l'invite de commandes sera représentée par `$`. La sortie éventuelle sera affichée à la ligne.

```
$ echo 1
1
```

Dans le cas d'une simple commande au sein d'un paragraphe, l'invite et le résultat ne seront pas affichés, par exemple **echo 1**.

De même, les noms de fichiers, modules, noms de classe ou de fonction, ou tout autre élément du langage, seront affiché en police télétype, comme `script.py`.

Lorsque des variables sont utilisées sans avoir été définies explicitement, il est sous-entendu que l'utilisateur doit le faire lui-même. Par exemple si on écrit

```
print a + 1
```

on part du principe que la variable `a` a été définie auparavant, par exemple comme `a = 2`. L'utilisateur devra prendre garde à ce que les types ne soient pas incompatibles (dans l'exemple ci-dessus, il est vital que `a` soit un nombre).

4.2 Symboles

J'utiliserai parfois certains symboles afin de simplifier les notations. Ces symboles sont les suivants :
– `|` est équivalent à OU, il indique donc une alternative, (`a|b` signifie que l'on peut écrire `a` ou `b`) ;
– `*` remplace un ou plusieurs caractères quelconques ;
– `[]` indique une option.
Ils ne seront utilisés que lorsque aucune confusion avec un autre élément du langage Python ne sera possible.

4.3 Touches

Lorsqu'il sera nécessaire d'appuyer sur une touche, cela se indiqué ainsi : `A`. Un `+` entre deux touches indiquent qu'il faut appuyer sur les deux touches en même temps : `Ctrl + A`.

4.4 Informations

Des encadrés peuvent donner des informations complémentaires :
– Remarque : un ajout à propos d'un sujet évoqué, moins important que le reste du sujet.
– Mise en garde : une remarque particulièrement importante.

Note : Le titre du cadre changera en fonction du sujet.

Une référence à une *PEP* (Python Enhancement Proposals) sera faite de manière très similaire à une information.

Voir aussi :

PEP 0 Index of Python Enhancement Proposals

4.5 Exercices

Les solutions aux exercices sont triées par partie puis par chapitre. Le nom du fichier correspond au numéro de l'exercice. Dans les cas où plusieurs algorithmes sont proposés, une lettre (a, b…) sera ajoutée après le numéro.

4.6 Divers

Les traductions en anglais des mots importants seront indiquées entre parenthèses, afin que le lecteur puisse consulter la littérature ou la documentation anglaises sans problèmes.

La version de Python utilisée dans ce livre est la 3.2, mais la majorité du contenu sera valable pour toute version 3.x (où x désigne un numéro quelconque de version).

Les présentations des modules n'incluent pas la description de toutes leurs fonctionnalités : l'objectif est de les présenter et de donner une idée de quel module utiliser en fonction du contexte. Pour une utilisation plus poussée, il faudra se reporter à la documentation.

Les exemples fournis seront uniquement écrits en anglais, pour la simple et bonne raison que la quasi totalité des projets sont en anglais, et qu'il s'agit d'une habitude à prendre (de plus, il faut s'abstraire de la langue utilisée et concentrer nos efforts sur la logique). L'autre raison sera détaillée dans le cœur du livre et concerne le fait qu'il est impossible d'utiliser des caractères UTF-8 (lettres accentuées, par exemple) et que cela nuit à la lecture du code.

De plus, je n'offrirai pas de comparaisons détaillées de Python par rapport aux autres langages.

Deuxième partie

Introduction

CHAPITRE 5

Langages de programmation

Les langages de programmation se distinguent par de nombreuses caractéristiques, qui doivent être prises en compte lors du choix du langage.

5.1 Interprété et compilé

Dans un langage compilé, le code source n'est pas directement lisible par la machine : il est nécessaire de le compiler pour le transformer en langage machine. On obtient alors un exécutable qui constitue le programme. Notons qu'une vérification de la syntaxe est faite lors de la compilation.

L'exécutable compilé dépend de la plateforme (c'est à dire du système d'exploitation), et il est nécessaire de recompiler le programme à la moindre modification.

Au contraire, le code source d'un langage interprété est transformé à la volée par un interpréteur en langage machine, ce qui explique la plus grande *portabilité* de ces langages par rapport aux langages compilés.

Toutefois il est nécessaire que l'interpréteur soit présent sur chaque machine, sans quoi le programme ne pourra pas fonctionner.

D'une manière intermédiaire, il existe les langages semi-interprétés, dont le code source est compilé en *bytecode*. C'est ce bytecode qui sera interprété, plus rapidement que si le code source ne l'avait directement été lui-même.

Exemples :
– interprété : PHP, Basic ;
– compilé : C, C++ ;
– semi-interprété : Java, Python.

5.2 Haut niveau et bas niveau

Le niveau d'un langage représente son degré d'abstraction : plus le niveau est haut, plus le langage est abstrait et compréhensible par l'humain.

Exemples :
– haut niveau : Basic, Python ;
– bas niveau : Assembler.

5.3 Typage

Il est nécessaire de déclarer manuellement le type de chaque variable dans les langages à typage statique. Par exemple en C, on aurait `int number = 1` (définition de la variable `number`, qui est de type "entier" et qui se voit attribuer la valeur `1`). De plus, dans un tel langage, le type d'une variable ne peut changer de lui-même en fonction du contexte.

Au contraire, dans les langages à typage dynamique, il n'est pas nécessaire de déclarer le type d'une variable. En Python, on pourra écrire `number = 1`, ce qui aura le même effet que le code plus haut en C, et, plus loin, `number = 'c'` après la dernière commande (`number` est cette fois-ci de type "chaine de caractères" et possède la valeur `c`), ce qui serait interdit en C.

Dans un langage à typage fort, il est impossible d'utiliser un type à la place d'un autre sans convertir explicitement l'objet. Au contraire, lorsque le typage est faible, un type peut aisément se substituer à un autre.

Par exemple en PHP (langage à typage faible et dynamique), on a

```
echo '1' + 1;
```

La sortie de ce programme sera '2'. Cela montre bien que l'on peut sommer une chaine de caractères et un nombre [1] : PHP convertit de lui-même la chaine de caractères en nombre car il juge cela adapté ici ; un tel programme en C ou en Python ne fonctionnerait pas.

Exemples :
– typage statique : C, Java ;
– typage dynamique : Python, PHP ;
– typage fort : Python ;
– typage faible : PHP.

1. Voir le Chapitre *Types de base* pour plus d'informations sur les types.

CHAPITRE 6

Paradigmes

Il existe différents paradigmes de programmation, c'est à dire d'approches des problèmes.

Un même langage peut être utilisé selon différents paradigmes, cela dépend essentiellement de l'objectif final.

Ci-après suit une liste, non exhaustive, des différents paradigmes qui existent.

6.1 Programmation impérative

En programmation impérative, le code est exécuté de manière presque linéaire : des boucles et des conditions permettent de contrôler le flux d'instructions.

Exemples : Assembler.

6.2 Programmation procédurale

Le code est est plus modulaire car structuré en fonctions ou procédures qui sont appelées à d'autres endroits.

6.3 Programmation orientée objet

Il s'agit d'un paradigme qui consiste à considérer chaque donnée manipulée comme un objet, définis par ses attributs (données le constituant) et ses méthodes (utilisées pour le manipuler).

Ce paradigme, au coeur de la programmation Python, sera détaillé dans le chapitre suivant.

Exemples : C++, Java, Python…

6.4 Programmation fonctionnelle

Exemples : Lisp, Scheme, Haskell, OCaml, Erlang…

CHAPITRE 7

Introduction à l'algorithmique

7.1 Structure

7.1.1 Grammaire

Il faut distinguer en première approche les instructions des expressions :

– Une *expression* est une combinaison d'opérateurs, de variables et de valeurs qui sera interprétée et qui retournera une valeur (typée). L'on peut assimiler une expression avec ce que l'on écrirait dans une calculatrice. Toutefois, une expression n'indique pas ce qui doit être fait de cette valeur, il faut donc une instruction pour la traiter.
  Ainsi, une expression peut être remplacée par une valeur du même type qu'elle renvoie tout en laissant le programme grammaticalement correct.
  Il existe différents types d'expressions :
  – expression arithmétique : un simple calcul (`1+2`, `10/2`...) ;
  – expression booléenne : une valeur booléenne, ou encore une comparaison (`True`, `1<2`...) ;
  – une variable ;
  – une chaine de caractères (`'chaine'`) ;
  – l'appel à une fonction.
– Une *instruction* indique qu'une action ou une commande doit être effectuée.
  Là aussi il en existe différents types :
  – affichage (`print 1`) ;
  – affectation : définition d'une variable (`a = 1`) ;
  – condition (`if a > 1: print a`) ;
  – boucle (`while a > 1: print a`).

Parfois, le lien entre expression et instruction dans le langage est ténu.

7.1.2 Syntaxe

Lignes

Une ligne physique se termine par le caractère de fin de ligne (`LF`, ou `\n`, sous les systèmes UNIX, `CRLF`, ou `\r\n`, sous Windows).

Dans la plupart des langages, une instruction peut s'étendre sur plusieurs lignes physique [1] : dans ce cas, un caractère spécial (comme le point-virgule `;` en C ou en PHP) marque la fin d'une ligne.

1. Nous verrons que ceci n'est pas le cas en Python, sauf cas particuliers.

Bloc

On désigne par bloc tout ensemble d'instructions reliées logiquement. Par exemple une suite d'instructions exécutées dans une boucle.

Tokens

Un token est une unit lexicale (on pourrait traduire par lexème, en français), c'est à dire un ensemble de symboles (ponctuation, lettres et chiffres, mots particuliers...) reconnu et interprété d'une manière spécifique. Par exemple, dans de nombreux langages, le symbole `if` permet d'introduire une condition. Le morceau de code

CHAPITRE 8

Programmation orientée objet

8.1 Général

Un objet est dit immuable (“imutable” en anglais) s’il ne peut pas être modifié une fois qu’il a été créé.

Un objet est dit appelable (*callable* en anglais) s’il est possible d’y faire appel pour l’utiliser. Cela revient à dire que l’on doit faire suivre son nom de parenthèses pour l’appeler.

8.2 Héritage

L’héritage consiste à *dériver* une ou plusieurs classes préexistantes afin d’en créer une nouvelle qui possèdera les attributs et méthodes des classes parentes (éventuellement modifiés) plus des attributs et méthodes nouveaux.

CHAPITRE 9

Concepts

9.1 Espaces de noms

9.2 Ramasse-miettes

Le *ramasse-miettes* est un mécanisme permettant de supprimer les variables qui ne sont plus utilisées par le programme.

Cela permet à l'utilisateur de ne pas (ou très peu) avoir à s'inquiéter de la gestion de la mémoire.

Python dispose d'un ramasse-miettes, Java aussi par exemple.

CHAPITRE 10

Méthodologie

10.1 Organisation

En programmation comme en toute chose que l'on crée de manière raisonnée, il peut être nécessaire d'agir avec méthode et rigueur, pour de multiples raisons : éviter de se disperser, gagner du temps, travailler efficacement…

Ainsi il convient d'adopter des conventions lors de l'écriture du code : elles permettent de s'y retrouver plus aisément lorsque l'on revient sur notre code plusieurs mois plus tard, d'adopter un style homogène lorsque plusieurs personnes travaillent sur un même projet (et ainsi de faciliter l'insertion des arrivants)… La communauté Python a d'ailleurs établi ses propres conventions (voir la section *Conventions et style*).

De même, il est important d'ajouter de nombreux commentaires pour accompagner le code, et ce pour plusieurs raisons :
– cela facilite la compréhension du code par les autres membres du projet ;
– cela permet de reprendre plus aisément un code mis de côté quelques temps.

10.2 Erreurs

Il est dit qu'un programmeur passe la majorité de son temps non pas à écrire du code, mais à le débugger. Il est possible de réduire ce temps en étant rigoureux, mais aussi en sachant adopter le bon comportement.

Ainsi, il est important de savoir analyser et comprendre les erreurs qui surviennent lors de l'exécution d'un programme (et en cela Python est d'une grande aide), la manière la plus triviale étant de comprendre le mode de fonctionnement interne du langage (sa logique) et *a fortiori* les détails de l'algorithme utilisé (car un code qui fonctionne un temps sans que l'on ne comprenne pourquoi sera très difficile à corriger si cela devient nécessaire). La programmation demande donc, plus que de la rigueur, de la logique ; pour rappel, l'algorithmique relève des mathématiques. Pour cette même raison il est déconseillé de recopier un code trouvé quelque part et l'utiliser tel quel (attention, je ne parle pas des modules, qui sont eux créés dans cette optique).

10.3 Recherche

La programmation consiste en un véritable travail de recherche : recherche des erreurs, dans la documentation, sur internet… Il s'agit d'un élément au coeur de tout projet, nécessaire pour mener à bien ce dernier.

Le premier aspect de la recherche est la découverte de ce qui provoque des bugs, via différentes méthodes.

De plus, il est généralement fortement déconseillé de chercher à apprendre par coeur plus que les bases du langage — car la logique importe bien plus, comme je l'expliquais au-dessus — et, ce faisant, il devient primordial de savoir trouver efficacement les informations que l'on cherche (aussi bien pour réparer un bug que pour explorer un point spécifique du langage ou encore ne pas "réinventer la roue" [1]).

Je recommande vivement à chaque programmeur, même débutant, d'avoir sous la main la documentation (que ce soit site internet ou fichier...) du langage concerné ainsi que celle des extensions les plus utilisées. Souvent, un index ainsi qu'une table des matières détaillées permettent de trouver facilement ce que l'on cherche dans tel ou tel domaine.

Il est aussi important de savoir chercher les informations sur internet (et pour cela connaitre les fonctionnements des moteurs de recherche) lorsque les informations sont insuffisantes dans la documentation.

Enfin, il est toujours de bon gout d'expérimenter, c'est à dire ne pas hésiter à modifier les exemples proposés, observer ce qu'ils donnent et, dès que le résultat n'est pas celui escompté, rechercher pourquoi et comment corriger le problème.

10.4 Autres conseils

Il est toujours conseillé de réfléchir à notre programme : savoir ce qu'il doit faire, comment, etc., et de ne pas se lancer directement dans la programmation. Cela permet d'avoir une idée d'ensemble du programme et de mieux le construire, comme je le conseillais dans la première section.

De plus, il est important de prévoir *tout* ce qui peut arriver dans un programme, et de traiter tous les cas imaginables, même les plus improbables, afin de ne pas avoir de surprises (par exemple, il est conseillé de toujours prévoir un choix par défaut lorsque l'on demande quelque chose à l'utilisateur). C'est d'ailleurs pour cette raison qu'il est recommandé de tester les programmes, afin de découvrir des comportements étranges.

1. "Not Invented Here", NIH, en anglais

Troisième partie

Le langage

CHAPITRE 11

Présentation

Python est un langage :
– semi-interprété ;
– de haut niveau ;
– orienté objet.
Il est utilisé dans de nombreux domaines, aussi bien pour concevoir des sites internet, des applications de bureautique, scientifiques, ou encore des scripts d'administration de systèmes…

Contrairement à d'autres langages, tel que Perl, où il existe de nombreuses manières de faire la même chose [1], une seule voie est privilégiée en Python. Ceci permet aux divers relecteurs de ne pas se retrouver désemparé face à une suite d'instructions ésotériques, impossibles à comprendre si l'on a pas lu à ce sujet.

Il n'y aura pas beaucoup d'exercices dans les premiers chapitres, puisqu'il s'agit d'apprendre les bases du langage, de s'habituer avec la syntaxe et les concepts. Toutefois, il est important de faire vos propres tests.

1. "There's More Than One Way To Do It", un adage Perl.

CHAPITRE 12

Installation et initialisation

12.1 Installation de Python

12.1.1 UNIX

Dans la plupart des cas, Python est préinstallé sur les systèmes UNIX (MacOS compris). Toutefois, la version installée est peut-être trop ancienne, et en ce cas il faut installer une plus récente.

Afin de vérifier la version installée, tapez dans votre terminal [1] :

```
$ python -V
```

La sortie de la commande devrait être quelque chose comme :

```
Python 2.7.2
```

Ou encore :

```
Python 3.2.1
```

Si la version est inférieure à la 3.0, alors il faut en installer une plus récente. Il faut être prudent quant à la coexistence de plusieurs versions de Python qui peut entrainer des incompatibilités.

Linux

La démarche la plus simple sous Linux est d'installer une version depuis les dépôts. Pour ce faire, utiliser votre gestionnaire de logiciels (*Synaptic* ou `apt-get` sous *Debian* par exemple) pour chercher la dernière version 3 disponible. Voici quelques exemples d'installation :
– *Debian* et dérivés (dont *Ubuntu*) : `sudo apt-get install python3`
– *ArchLinux* : `pacman -S python3`
– *Fedora* : `yum install python3`
Il peut arriver que la version 3 ne soit pas disponible dans les dépôts, en ce cas il faut compiler soi-même une version de Python depuis les sources (trouvées sur http ://www.python.org/download/), mais ceci dépasse le cadre de ce livre. Avant d'en arriver là, vérifier si le nom du paquet n'est pas légèrement différent (par exemple `python3.x` au lieu de `python3`).

1. Il est accessible dans *Applications* → *Utilitaires* → *Terminal* sous MacOS.

MacOS

L'installation à l'aide de `brew` doit fonctionner :

```
brew info python3
```

Sinon il peut être nécessaire là aussi de compiler les sources.

En autre recours, il est possible de consulter cette page : http ://www.python.org/download/mac/.

12.1.2 Windows

Python n'est presque jamais installé sous Windows, il faut donc le faire soi-même après l'avoir téléchargé : http ://www.python.org/download/. Choisir « Python 3.x Windows x86 MSI Installer (Windows binary — does not include source) ». [2]

Il est conseillé de l'installer dans un dossier à la racine de `C:`, par exemple dans `C:\python`.

Il convient ensuite d'ajouter Python au PATH de Windows, ce qui permettra de lancer les scripts directement par un double-clic. Pour ce faire :

1. Clic droit sur "Poste de travail".
2. Propriétés.
3. Onglet "Avancé".
4. Bouton "Variables d'environnement".
5. Dans la section "Variables systèmes", sélection "Path" et appuyer sur "Modifier".
6. Ajouter un point-virgule, suivi du chemin vers l'exécutable Python.
7. Par exemple, si vous n'avez rien changé : C :\Python31\python.exe
8. Il faut redémarrer pour que le changement prenne effet.

12.2 Lancement et utilisation

Il existe deux principales manières d'utiliser Python :
– en mode interactif ;
– en lançant un script.

12.2.1 Mode interactif

Bien qu'il puisse apparaitre peu utile au premier abord, le mode interactif sera d'une grande aide pendant toute votre vie de développeur Python : il permet de se familiariser avec la syntaxe au début, d'utiliser l'aide interactive sur les commandes... De plus, à moins que vous n'utilisiez des commandes qui affectent les fichiers du système, tout est réinitialisé dès que vous quittez le mode interactif, donc rien de grave ne peut arriver.

Dans un terminal UNIX ou DOS, il suffit de taper `python` (ou `python3` si la version 3 n'est pas celle installée par défaut [3]) pour lancer Python en mode interactif. L'invite principale `>>>` apparait alors (rappelons que l'invite est indiquée dans les exemples, mais il ne faut pas la recopier, car elle n'est là que pour signifier que l'exemple a lieu dans le mode interactif). Lorsque l'on écrit plusieurs commandes reliées ensemble (condition, boucle), l'invite principale est remplacée par l'invite secondaire `...`. Voici un exemple :

2. Vous pouvez aussi télécharger « Python 3.x Windows AMD64 installer » si vous avez un processeur 64 bits.
3. *A fortiori*, il est possible d'utiliser `pythonx.y` pour lancer la version x.y (à condition qu'elle soit installée, bien entendu).

```
>>> 1+1
2
>>> if 2 > 1:
...     print('2 est plus grand que 1.')
2 est plus grand que 1.
```

Note : Sous Windows, pour ouvrir le terminal DOS, il faut faire chercher *Exécuter* (accessible dans le menu *Démarrer* jusqu'à Windows XP) puis taper `cmd` et valider. Sur les Windows plus récents, le menu *Exécuter'* est caché par défaut : pour le faire apparaitre, il est nécessaire de cocher la case *Commande Exécuter* dans *Propriétés* → *Menu Démarrer* → *Personnaliser*. On peut aussi y accéder par le raccourci `Windows + R`.

Dans tous les cas, il est possible de lancer un terminal Python via les programmes.

Warning : Il sera fortement conseillé de préférer IPython (détails en *annexe*) au mode interactif de base, lorsque celui-ci supportera enfin Python 3.

Ainsi, on écrira une commande directement à la suite de l'invite, et on validera avec la touche `Entrée`. Après chaque commande, l'éventuel résultat est affiché, puis une nouvelle invite apparait.

Une variable peut directement être afficher, sans nécessairement utiliser la fonction `print()`, pour ce faire, il suffit d'écrire le nom de la variable :

```
>>> a = 1
>>> a
1
```

De plus, une variable spéciale, l'underscore _, est disponible en mode interactif : sa valeur vaut celle de la dernière expression.

```
>>> 1 + 1
2
>>> _
2
>>> _ + 1
3
```

Note : La combinaison `Ctrl+D` permet de quitter le mode interactif python. On peut aussi utiliser la fonction `exit()`.

La fonction `help()`, définie seulement dans le mode interactif, permet d'afficher des informations sur un objet (son "docstring" plus précisément) : celles-ci comprennent les arguments qu'il faut fournir à l'objet, une courte description ainsi que la liste des méthodes et des attributs (accompagnés eux aussi d'une description) :

```
>>> help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file: a file-like object (stream); defaults to the current sys.stdout.
    sep:  string inserted between values, default a space.
    end:  string appended after the last value, default a newline.
```

Si on ne lui donne aucun argument, alors la fonction `help()` lance l'aide interactive, et l'on peut demander des informations sur plusieurs objets à la suite. On peut quitter ce mode en écrivant `q` ou `quit`.

12.2.2 Fichier

Généralement, l'on écrit le code Python dans un fichier (portant souvent l'extension ".py"), que l'on peut lancer depuis le terminal avec la commande `python script.py`. Un tel fichier peut être édité avec n'importe quel éditeur de texte (on se reportera à l'*annexe sur les éditeurs* pour plus de détails et quelques conseils).

Il est aussi possible de lancer un script par un double-clic si Python est dans le PATH de Windows ou, dans le cas des systèmes UNIX (si le fichier a été rendu exécutable), si la première ligne du fichier est

```
#! /usr/bin/python3
```

Le système utilisera la version la plus récente de Python 3 (si on avait indiqué seulement `python`, alors le système choisit la version par défaut, qui est souvent une 2.x).

Finalement, la solution la plus simple (au moins pour des programmes peu étendus, ainsi que pour tester) pour lancer un programme Python consiste à utiliser la ligne de commandes :

```
python3 name.py
```

après s'être placé (avec `cd`, par exemple) dans le répertoire contenant le fichier à lancer.

CHAPITRE 13

Bases et structure

Dans ce chapitre, nous allons étudier l'aspect structurel général du Python, à savoir comment on présente le code, quels sont les mots et les symboles importants... Le but est de donner une idée de la manière dont Python est construit et dont les programmes sont analysés. Ne soyez pas surpris si nous n'avons pas encore vu le sens de certaines expressions, elles ne sont là que pour donner un aperçu visuel et pour permettre de lire un programme Python : l'essentiel est de savoir reconnaitre les divers symboles et l'organisation logique, le reste viendra avec l'habitude.

Note : Même si l'aspect technique de ce chapitre peut rebuter, il est important au moins d'y jeter un œil afin d'avoir une idée générale de ce que l'on peut faire ou pas faire (les deux sections sur les variables sont vitales). On pourra souhaiter relire ce chapitre après s'être habitué au langage à travers les chapitres suivants, afin de mieux saisir certains points.

On peut décomposer les tokens en deux types :
– ceux responsables de la structure générale, que l'on peut assimiler à l'aspect visuel : indentation, nouvelle ligne...
– les autres, qui sont plus proches du langage : variables, mots-clés, délimiteurs...

13.1 Tokens de structure

13.1.1 Lignes

Il faut distinguer les lignes physiques (c'est à dire les lignes affichées à l'écran) des lignes logiques (telles qu'elles sont reconnues par Python). Généralement, une ligne physique correspond à une ligne logique, et une instruction Python ne peut pas s'étendre sur plusieurs lignes. Toutefois, il est possible de créer une ligne logique s'étendant sur plusieurs lignes réelles de deux manières :
– explicitement : il faut indiquer à l'interpréteur qu'une instruction s'étend sur plusieurs lignes en ajoutant un antislash \ à la fin de la ligne. La ligne qui suit peut être indentée de n'importe quelle manière. Il est quand même conseillé d'adopter une certaine logique pour facilier la lecture.

```
>>> if a > 1 and b > 5 \
...         and c < 3:
...     print(1)
```

– implicitement : une expression entourée de parenthèses (ou de crochets, ou encore d'accolades) peut s'étendre sur plusieurs lignes. De même, pour les dictionnaires ou les listes.

```
>>> if (a > 1 and b > 5
...         and c < 3):
...     print(1)
```

Note : Il est souvent plus lisible d'utiliser des parenthèses qu'un antislash. Dans les deux cas, l'indentation de la deuxième ligne est arbitraire :

```
>>> if (a > 1 and b > 5
... and c < 3):
...     print(1)
```

On préfèrera malgré tout le premier exemple qui est beaucoup plus lisible.

Plusieurs instructions peuvent être placées sur une même ligne à condition d'être séparées par un point-virgule `;`, mais cela nuit grandement à la lisibilité : on réservera donc cette écriture au mode interactif :

```
>>> print(1); print(2)
1
2
```

Enfin, une ligne vierge ou contenant uniquement des espaces ou tabulations est ignorée par l'interpréteur. Elles peuvent donc être insérées selon le bon vouloir du programmeur afin d'aérer le code et de séparer les blocs logiques.

Note : `print()` est une fonction permettant d'afficher du texte. Son utilisation sera détaillée dans le chapitre *Entrées et sorties*.

Pour le lecteur curieux, on notera que le token permettant d'indiquer la fin d'une ligne logique est nommé `NEWLINE`.

13.1.2 Blocs et indentation

En Python, les blocs d'instructions sont délimités par l'indentation. Pour cette raison, il n'est pas possible d'indenter librement les lignes comme on peut le faire dans d'autres langages.

```
a = 1
if a > 2:
    b = a + 1
    print(b)
print(a)
```

Les lignes 3 et 4 ne seront exécutées que si la condition est vraie (ici ce n'est pas le cas), tandis que la dernière, qui n'est pas indentée, n'est pas inclue dans la condition.

Voici un exemple de mauvaise indentation :

```
>>>     print(1)
  File "<stdin>", line 1
    print(1)
    ^
IndentationError: unexpected indent
```

Reprenant l'exemple précédent, mais avec une légère erreur d'indentation :

```
>>> if a > 2:
...     b = a + 1
...      print(b)
  File "<stdin>", line 3
    print(b)
    ^
IndentationError: unexpected indent
```

L'on rajoute facilement un espace en début de ligne, espace qui sera compté comme une indentation et causera donc une erreur. Il faut y prendre garde.

Les indentations et désindentations successives sont représentées par les tokens `INDENT` et `DEDENT`. Ainsi, ce sont les indentations relatives qui sont importantes, et il n'est pas nécessaire qu'elles soient consistantes dans tout le code :

```
>>> if a > 2:
...     print(1)
... else:
...         print(2)
...
1
```

Mais par pitié, ne faites *jamais* ça !

13.1.3 Commentaires

Un commentaire est introduit par le signe `#` et s'étend jusqu'à la fin de la ligne. Il peut être ajouté à la fin d'une expression

```
>>> # commentaire
>>> # print(1)
>>> print(1) # un autre commentaire
1
```

Les commentaires sont ignorés au moment de l'analyse du code et ne sont donc pas considérés comme des tokens.

13.2 Autres tokens

13.2.1 Variables

Une variable permet de stocker une donnée qu'il sera possible de réutiliser plus tard. L'aspect dynamique des variables permet d'expliquer les interactions possibles avec un programme [1].

Le nom d'une variable :
– doit contenir uniquement des caractères alphanumériques (lettres non-ASCII comprises), ou des underscores ;
– doit commencer obligatoirement par un caractère alphabétique ou un underscore ;
– est sensible à la casse (cela signifie que `var` est différent de `Var` ou encore `VAR`) ;
– ne peut être nommée de la même manière qu'un mot-clé (voir section *Mots-clés*).

Note : La possibilité d'utiliser des caractères non-ASCII est apparue en Python 3. Toutefois, malgré la possibilité d'écrire ainsi le programme dans n'importe quelle langue (français, grec, chinois...), il est vivement déconseillé de le faire : pour qu'un programme vive, il doit y avoir des échanges, et l'anglais est (hélas) la langue qui est généralement comprise par tous.

1. Par exemple, une page internet en HTML pur ne saurait être dynamique car ce langage ne possède pas de variables.

Voir aussi :

PEP 3131

Warning : Même si cela est possible, il est fortement déconseillé de renommer une fonction built-in :

```
>>> print = 2
>>> print('test')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
```

Dans la second expression, Python essaie d'appeler le chiffre `2` avec l'argument `'test'`. Or un nombre n'a pas d'attribut `call` (c'est à dire qu'il ne peut pas se comporter comme une fonction), d'où l'erreur.

Utiliser un identifiant invalide lèvera une exception `SyntaxError`.

```
>>> a@ = 1
Traceback (most recent call last):
  File "<stdin>", line 1
    a@ = 1
     ^
SyntaxError: invalid syntax
>>> 2abc = 1
Traceback (most recent call last):
  File "<stdin>", line 1
    2abc = 1
       ^
SyntaxError: invalid syntax
```

Note : Il existe certaines conventions de nommage pour les variables (ainsi que pour les classes, fonctions, etc). Voir la section *Conventions et style*. En première approche, il est conseillé d'utiliser uniquement des minuscules et des underscores, et d'essayer de faire des noms aussi courts que possible (mais qui conservent un sens).

Il existe certaines catégories d'identifiants spéciaux, reconnaissables à un schéma particulier :
– `_*` est utilisé pour indiquer les attributs protégés. Ces derniers ne seront pas importés par `from module import *`, ni directement accessible par la notation pointée.
– `__*` est utilisé pour indiquer les attributs privés.
– `__*__` est utilisé par certains noms définis par le système. Une application ne devrait pas avoir à utiliser des identifiants de cette forme.
Nous les reverrons en temps voulu.

13.2.2 Littéraux

Les littéraux sont des valeurs constantes de certains types prédéfinis :
– les chaines de caractères et autres types similaires (encadrés par des guillemets) ;
– les nombres (entiers, flottants et imaginaires).
Voici quelques exemples : `'a'`, `'abc'`, `1`, `2.3`...

13.2.3 Mots-clés

Les mots-clés introduisent une instruction et ils servent à indiquer une action spéciale ou à définir une variable très spéciale. Contrairement aux fonctions, leur argument n'est pas entouré de parenthèses.

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

L'utilisation de ces mots-clés sera détaillée au cours du livre.

On parle aussi de mots réservés, car on n'a pas le droit de les utiliser comme nom de variables :

```
>>> for = 2
  File "<stdin>", line 1
    for = 2
        ^
SyntaxError: invalid syntax
```

Note : Dans les versions 2.x, `print` et `exec` sont des mots-clés. Ne vous étonnez donc pas de les voir être utilisés sans parenthèses dans des programmes écrits dans une ancienne version.

13.2.4 Opérateurs

Un opérateur permet d'exécuter, comme son nom l'indique, une opération particulière. Les valeurs auxquelles ils sont appliqués sont appelées opérandes.

Il existe de nombreux opérateurs, listés dans le tableau *Liste des opérateurs*. Leurs effets sont différents selon les types avec lesquels ils sont utilisés, par exemple :

```
>>> 1 + 1
2
>>> 'a' + 'b'
'ab'
```

Leurs utilisations seront donc détaillées dans les sections associées à chaque type. Nous verrons aussi comment surcharger les opérateurs, c'est à dire les utiliser avec de nouveaux objets.

```
&       |       ^       ~
<<      >>      ==      !=
<       >       <=      >=
+       -       *       **
/       //      %
```

Il est nécessaire de faire attention à l'ordre des opérateurs, car ils ont des priorités différentes lors de l'évaluation [2].

Note : Il est de coutume de placer un espace autour de chaque opérateur ; comme toute convention, celle-ci n'est bonne à respecter que si elle ne nuit pas à la lisibilité : par exemple, on écrira plutôt `(2+1)*4` que `(2 + 1) * 4`.

---

2. Au même titre que les opérateurs en mathématiques.

13.2.5 Délimiteurs

Voici la liste des délimiteurs :

```
(       )       [       ]       {       }
,       :       .       ;       @       =
+=      -=      *=      /=      //=     %=
&=      |=      ^=      >>=     <<=     **=
```

Notons que le point `.` sera aussi utilisé comme séparateur décimal.

13.3 Variables

Dans cette section nous allons voir plus en détails l'utilisation des variables.

13.3.1 Affectation

On parle aussi d'assignation. L'affectation se fait grâce au signe =. Elle permet d'attribuer une valeur à variable. Il est bien entendu possible de changer le contenu d'une variable en lui affectant une nouvelle valeur.

```
>>> a = 1
>>> a
1
>>> a = 'var'
>>> a
'var'
```

Lors de l'affectation, l'interpréteur effectue plusieurs actions :
– il crée et stocke l'objet (et détermine son type) associé à la variable s'il n'existe pas encore ;
– il stocke le nom de la variable ;
– il effectue un lien (on parle de référence) entre l'objet et le nom de la variable ;
– il attribue dynamiquement un type à l'objet.
Toutes les informations sont stockées dans la mémoire de l'ordinateur.

> **Warning :** Il ne faut pas confondre l'opérateur d'affectation avec celui d'égalité (==).

13.3.2 Affectation calculée

Il est possible de raccourcir l'instruction `a = a+1` [3] en `a += 1`. De même, cela est possible avec tous les autres opérateurs binaires.

```
>>> a = 0
>>> a += 2
>>> a
2
>>> a *= 8
>>> a
16
>>> a %= 3 + 2
```

3. En mathématiques, cette notation n'a bien entendu aucun sens. Toutefois, comme il est précisé, plus haut, il ne s'agit pas d'une égalité mais d'une affectation.

```
>>> a
1
```

Au cas où la dernière affectation serait difficile à comprendre, il faut la lire ainsi : `a = a % (3+2)`. Bien que l'on ait utilisé des chiffres dans cet exemple, l'affectation calculée est définie pour tous les opérateurs et reste valable quels que soient les types d'objets manipulés :

```
>>> s = 'ha'
>>> s *= 5
>>> s
'hahahahaha'
```

Note : On parle d'incrémentation dans le cas où l'on ajoute 1 par cette méthode, et de décrémentation dans le cas où l'on retranche 1.

Les notations du type `a++` que l'on peut voir dans d'autres langages n'existent pas en Python.

13.3.3 Références

Lorsque l'on assigne le même objet à différentes variables, cet objet n'est pas créé à nouveau à chaque fois : une référence est créée. Elle peut être vue comme un lien entre les noms, et le contenu. Tant qu'au moins un lien existe, l'objet perdure. Il ne sera supprimé que lorsque la dernière référence disparait (grâce au ramasse-miettes).

Par exemple, le code suivant ne créera qu'un seul objet, mais trois références :

```
>>> x = 1
>>> y = 1
>>> z = 1
```

Il est d'ailleurs possible de vérifier que l'objet est bien unique grâce au mot-clé `is` (qui sera détaillé plus loin) :

```
>>> x is y
True
```

13.3.4 Affichage d'une variable

Nous prendrons un peu d'avance afin d'aborder le point essentiel qu'est l'affichage d'une variable dans le mode interactif ou dans le terminal pour un programme lancé depuis la ligne de commande. Nous avons déjà vu la première méthode, uniquement en mode interactif, qui consiste à écrire le nom de la variable au prompt :

```
>>> var = 'a'
>>> var
'a'
```

La seconde solution est d'utiliser la fonction `print()`.

```
>>> print(var)
a
>>> print('chaine')
chaine
```

Elle admet une grande diversité d'arguments, que nous étudierons plus tard. Notons simplement que si plusieurs arguments sont fournis, alors ils seront tous affichés sur la même ligne, séparés par un espace :

```
>>> print('Nous sommes le', 2, 'décembre.')
```

Note : Remarquez que dans le premier cas, `'a'` est affiché, tandis que dans le second l'on obtient `a`. En fait, `print` affiche la valeur de l'objet, tandis que l'autre méthode affiche la représentation de l'objet. Nous verrons cela en détails plus loin.

13.3.5 Affectations multiples

Il est possible d'affecter plusieurs valeurs à plusieurs variables en une seule ligne :

```
>>> a, b, c = 1, 2, 'var'
>>> a
1
>>> b
2
>>> c
'var'
```

On peut aussi affecter la même valeur à plusieurs variables d'un seul coup de la manière suivante :

```
>>> a = b = 1
>>> a, b
(1, 1)
```

Toutefois cette écriture peut avoir des effets pervers, car Python crée deux références d'un même objet, et non deux objets différents : ainsi, si on modifie l'objet à travers une des deux variables, alors la modification sera répercutée dans la deuxième. Pour illustrer ce fait, créons deux listes qui contiennent à l'origine la séquence de chiffre `1, 2, 3`, et nous cherchons ensuite à remplacer le dernier chiffre de la seconde liste uniquement :

```
>>> l1 = l2 = [1, 2, 3]
>>> l2[2] = 5
>>> l2
[1, 2, 5]
>>> l1
[1, 2, 5]
```

Ainsi que nous le voyons, la valeur de la première liste a aussi changé ! On peut vérifier que les deux listes ne sont pas deux objets distincts :

```
>>> l1 is l2
True
```

Il existe plusieurs solutions à ce problème, que nous verrons dans le chapitre sur les listes. Nous esquissons une première solution dans la note.

Note : Reprenons l'exemple précédent où l'on a assigné la valeur `1` à `a` et `b`, et changeons la valeur de `b` :

```
>>> a = b = 1
>>> b = 2
>>> a, b
(1, 2)
```

Pourquoi `a` n'a-t-il pas pris la valeur `2` lui aussi ? La raison est que nous avons affecté une *nouvelle* valeur à la variable `b`, et donc un nouvel objet, ce qui ne modifie en rien l'objet précédent. Ainsi, une première méthode pour contourner le problème précédent consiste à recréer une liste et à l'assigner à `b`.

13.3.6 Échange de valeurs

Parfois l'on souhaite échanger les valeurs de deux variables. Là où d'autres langages obligent le programmeur à utiliser une variable intermédiaire, Python offre une solution élégante :

```
>>> x, y = y, x
```

Ceci est une conséquence directe de l'affectation multiples vue juste avant. Ainsi, il est de même possible d'échanger les valeurs de plusieurs variables :

```
>>> c, a, d, b = a, b, c, d
```

Vérifions-le sur un exemple particulier :

```
>>> x, y = 1, 2
>>> x, y = y, x
>>> print(x, y)
2 1
```

13.3.7 Suppression d'une variable

Pour supprimer une variable, nous utilisons le mot-clé `del`, qui accepte un ou plusieurs arguments. Dans l'exemple qui suit, nous supposons que les variables ont été déclarées précédemment.

```
>>> del a
>>> del b, c, d
>>> del a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

Nous remarquons qu'une exception est levée si nous essayons de supprimer une variable qui n'existe pas (ou plus).

13.4 Résumé

Dans ce chapitre nous avons expliqué la construction d'un programme Python, en détaillant les structures et les identifiants reconnus par le langage.

Les différents tokens non liés à la structure sont les suivants :
– identifiants de variables ;
– les littéraux ;
– mots-clés ;
– opérateurs ;
– délimiteurs.
Les espaces permettent de séparer les tokens, mais ils n'en sont pas eux-mêmes.

Ensuite, nous avons détaillé l'utilisation des variables : création et gestion par l'ordinateur, suppression et affichage.

13.5 Exercices

1. Essayer de changer le contenu de deux variables sans utiliser l'affectation multiple.

CHAPITRE 14

Types de base

Nous allons maintenant étudier les principaux objets utilisés en Python, que nous appellerons des *types* : il s'agit d'objets relativement simples, comparés à ceux que l'on peut trouver ou construire dans des modules spécialisés, et ces types représentent des objets assez intuitifs :
– les nombres (entier, à virgule...) ;
– les listes, les chaines de caractères et d'autres séquences ;
– les booléens, l'élément nul...
Nous ne détaillerons pas toutes les subtilités ni les usages des types tout de suite, et nous nous contenterons de jeter un œil à leur définition, à leurs principales caractéristiques ainsi que leur comportement en fonction des opérateurs.

14.1 Introduction

Commençons par présenter deux fonctions que nous utiliserons continuellement :

type(*object*)
Retourne le type de `object` au moyen un objet de type.

isinstance(*object*, *classinfo*)
La variable `classinfo` peut être une classe ou un tuple de classes. La fonction retourne `True` si `object` est une instance d'une des classes de la liste ou d'une sous-classe.

Nous verrons quelques exemples explicites dans les sections qui suivent.

On préférera réserver l'usage de `type()` au mode interactif, quand on veut vérifier le type d'un objet : en effet, la fonction `isinstance()` est plus robuste car elle vérifie aussi si l'objet ne dérive pas d'une sous-classe des classes indiquées. De plus, comme elle retourne un booléen, la fonction est plus "propre" pour écrire un test. Comparer les écritures suivantes :

```
>>> type(1) == int
True
>>> isinstance(1, int) is True
True
```

On aurait même pu écrire omettre le `is True` comme la syntaxe est assez claire.

Note : `type()` retourne en fait l'objet de classe et permet donc de construire directement un objet du même type que son argument :

```
>>> type(1)('102')
102
```

qui a bien le même effet que :

```
>>> int('102')
102
```

Cette construction peut être utile si on veut convertir un objet dans le même type qu'un second, sans connaitre ce type à l'avance.

14.2 Nombres

14.2.1 Introduction

Cette section suppose que le lecteur possède quelques notions de mathématiques, même si certains aspects un peu plus avancés sont traités afin de viser l'exhaustivité.

Note : Tous les objets nombres dérivent de la classe `numbers.Number`.

14.2.2 Flottants

Les flottants, ou nombres réels, représentent les nombres à virgule. Ils peuvent être représentés de plusieurs manières :
– notation à virgule (ex. : `1.1`) : la partie entière est séparée de la partie décimale par un point [1] ;
– notation scientifique (ex. : `11e-1`) : un nombre est mis multiplié par la puissance de 10 indiqué après un `e` (minuscule ou majuscule) : mathématiquement, cela revient a écrire $11 \times 10^{-1} = 1.1$.
La fonction `float()` permet de convertir un nombre en flottant.

La précision de ces nombres dépend entièrement de la machine sur laquelle est exécutée Python, même si cela n'est pas affiché explicitement à l'écran.

```
>>> 1.1
1.1
>>> type(1.1)
<class 'float'>
>>> float(1)
1.0
>>> type(1.0)
<class 'float'>
>>> 2e-2
0.02
>>> type(2e-2)
<class 'float'>
```

Un nombre entier dont la partie décimale est explicitement représentée est considéré comme un nombre à virgule. Cela pouvait être utile dans les versions précédentes mais beaucoup moins dans la version 3.

Nous remarquons un effet de bord indésirable lorsque l'on entre `1.1` : il s'agit d'un problème commun à tous les langages dès que l'on utilise des nombres flottants. Ceci est dû à la manière dont les nombres sont gérés.

1. La notation des nombres à virgule diffère ici de celle habituellement adoptée en France.

Warning : Il faut faire très attention à utiliser le point et non la virgule pour les nombres décimaux : deux nombres séparés par une virgule seront considérés comme un tuple.

Warning : Les anciennes versions de Python peuvent afficher de manière incorrecte les nombres flottants à cause d'erreurs d'arrondis :

```
>>> 1.1
1.1000000000000001
```

14.2.3 Entiers

Il s'agit des nombres entiers ("integer" en anglais, abrégé en "int"), positifs ou négatifs. Leur taille a pour seule limite la mémoire virtuelle du système (là où les nombres d'autres langages sont limités, par exemple les `int` en C ne peuvent dépasser 32 bits).

La fonction `int()` permet de convertir un objet en nombre entier. Si on lui transmet un nombre flottant, alors elle ne garde que la partie entière.

```
>>> 1
1
>>> type(1)
<class 'int'>
>>> 23923 * 985
23564155
>>> 123456789987654321234567899876543212345678 9
123456789987654321234567899876543212345678 9
>>> int(3.5)
3
```

Une manière de s'assurer qu'un nombre est bien entier est d'utiliser l'expression `n == int(n)`, qui ne sera vraie que si `n` est un nombre entier à la base (même écrit sous forme de flottant) :

```
>>> a, b, c = 1, 1.0, 1.5
>>> a == int(a)
>>> b == int(b)
True
>>> c == int(c)
False
```

14.2.4 Complexes

Note : À moins que vous n'étudiez les sciences (et encore), cette section sera pour ainsi dire inutile et vous pouvez l'ignorer sans états d'âme.

Les nombres complexes (ou imaginaires) ont été inventés pour résoudre certains problèmes mathématiques. Depuis, ils sont utilisés dans de nombreux domaines (ingénierie, physique...). Un nombre complexe est de la forme `a + bj`[2], où `a` et `b` sont des réels [3] [4]. La forme utilisant un `J` est aussi acceptée.

On peut obtenir un nombre complexe grâce à la fonction `complex()`.

2. On a $j^2 = -1$.
3. `a` est appelé partie réelle et `b` est appelé partie imaginaire.
4. Les mathématiciens préfèrent utiliser la lettre i.

```
>>> c = 2+3j
>>> c
(2+3j)
>>> type(c)
<class 'complex'>
>>> 2.2j
2.2j
>>> 1+2J
(1+2j)
>>> complex(3.5)
(3.5+0j)
```

Les parties réelles et imaginaires d'un nombre complexe sont accessibles avec les attributs `real` et `imag` (qui renvoient des float). Enfin, le conjugué s'obtient avec la méthode `conjugate()`.

```
>>> c.real
2.0
>>> c.imag
3.0
>>> c.conjugate()
(2-3j)
```

Note : Il est impossible d'écrire simplement `j` car l'interpréteur le considère comme l'identifiant d'une variable. Le nombre purement imaginaire unité s'écrit donc `1j`.

14.2.5 Booléens

Il s'agit d'une classe dérivée de `int`.

Les booléens indiquent une valeur de vérité : il n'existe ainsi que deux valeurs possibles : vrai (1, qui correspond à l'objet `True`) ou faux (0, soit `False`). La conversion en booléen se fait avec la fonction `bool()`.

Nous reviendrons sur ces objets et leur utilisation dans le chapitre *Structures de contrôle*.

14.2.6 Opérations

La plupart des opérations numériques classiques (addition, soustraction, multiplication...) existent en Python et sont accessibles par les opérateurs idoines :
– `a + b` : addition de `a` et `b` ;
– `a - b` : soustraction de `b` à `a` ;
– `a * b` : multiplication de `a` et `b` ;
– `a / b` : division de `a` par `b` ;
– `a // b` : division entière de `a` par `b` ;
– `a % b` : reste de la division entière (modulo) de `a` par `b` ;
– `a**b` : mise de `a` à la puissance `b` ;
– `-a` : opposé de `a` (revient à multiplier par `-1`)[5] ;
– `+a` : positif de `a` (revient à multiplier par `1`).
Les comparaisons sont celles usuellement définies avec les nombres : un nombre est supérieur à un autre s'il est plus grand...

Ces opérations sont valables avec chacun des nombres définis précédemment et peuvent même être utilisées entre des nombres de types différents.

5. Ainsi, les nombres négatifs ne sont pas des nombres en eux-mêmes, mais plutôt considéré comme les opposés des nombres positifs.

Voici quelques exemples d'utilisation :

```
>>> 2 + 3.5
5.5
>>> 3 * 6
18
>>> 3 - 7
-4
>>> 7 / 2
3.5
>>> 7 // 2
3
>>> 7 % 2
1
>>> (2+3j) + 0.1
(2.1+3j)
>>> -+-1
1
```

Le dernier exemple montre bien qu'il s'agit d'opérateur unaire (même si ce genre d'opérations est peu intéressant). L'opération modulo `%` est extrêmement utile pour déterminer si un nombre `a` est multiple d'un autre `b` : en effet, si `a` est un multiple de `b`, alors `b` divise exactement `a` et le reste est nul. Ainsi, l'expression `a % b == 0` sera vraie :

```
>>> 9 % 3 == 0
True
```

car 9 est bien un multiple de 3.

Python sait reconnaitre un même nombre même s'il est admet plusieurs écritures différentes (donc même s'il est représenté par différents objets) :

```
>>> 1 == 1.0
True
>>> 1 == (1 + 0j)
True
```

Note : La coercition est une opération consistant à convertir de manière implicite le type d'un objet en un autre type lors d'une opération (grâce à l'usage implicite de la fonction built-in `coerce` et à la méthode spéciale de classe `__coerce__`), afin d'éviter au programmeur de devoir le faire manuellement. Ainsi, cela permet d'utiliser les opérations qui viennent d'être définies entre des nombres de types différents. Il s'agit d'une fonctionnalité dépréciée depuis Python 3, car elle contredisait l'idée du typage fort.

Ceci ne change pas réellement les habitudes du programmeur puisque, lors de la définition d'un nouveau type, il fallait et il faudra dans tous les cas prendre en charge explicitement les conversions.

14.3 Conteneurs et séquences

Les conteneurs sont des objets qui en contiennent d'autres. Une sous-catégorie très importante des conteneurs est celle des séquences : dans ces dernières, les objets contenus sont ordonnées selon un certain ordre. Dans la suite de cette section, nous allons étudier quelques séquences (listes, tuples et chaines de caractères), ainsi que les dictionnaires et les ensembles.

Dans une séquence, un objet est repéré par un indice ("index" en anglais), qui vaut 0 pour le premier élément de la séquence, et n-1 pour le dernier (n étant la longueur de celle-ci).

Warning : Il convient de faire très attention au fait que la numérotation des index commence à 0. Il s'agit d'une particularité en programmation, très commune.

Dans le cas d'un conteneur non ordonné, l'accès aux éléments — lorsqu'il est possible — se fait à l'aide d'une clé ("key") : il s'agit d'un identifiant, qui peut être n'importe quel objet non modifiable (une chaine, un nombre, un tuple...).

Les index et clés sont indiqués entre crochets immédiatement après l'objet.

```
sequence[key]
sequence[index]
```

L'accès avancé aux éléments, notamment avec le slicing, sera abordé plus loin.

14.3.1 Listes

Une liste est délimitée par des crochets, et chaque élément est séparé du précédent par une virgule. Il s'agit d'un conteneur dont les différents éléments peuvent être modifiés grâce à une simple affectation. L'ajout se fait avec la méthode `append()`. Finalement, la fonction `list()` permet de convertir un objet en liste.

```
>>> l = ['a', 'b', 'c']
>>> l
['a', 'b', 'c']
>>> type(l)
<class 'list'>
>>> l[0]
'a'
>>> l[2]
'c'
>>> l[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> l[1] = 'e'
>>> l
['a', 'e', 'c']
>>> l.append('z')
['a', 'e', 'c', 'z']
```

14.3.2 Tuples

Les tuples sont construits à partir de plusieurs éléments séparés par des virgules ; le tout peut éventuellement être délimité par des parenthèses. Les tuples sont très semblables aux listes, mis à part qu'ils sont immuables : comme conséquence, ils consomment moins de mémoire et ils sont à privilégier si l'on sait que l'on ne souhaite pas modifier la séquence d'objets.

La fonction `tuple()` permet de convertir en tuple.

```
>>> t1 = 'a', 'b', 'c', 'd'
>>> t1
('a', 'b', 'c', 'd')
>>> t = ('a', 'b', 'c', 'd')
>>> t[1]
'b'
>>> t[1] = 'e'
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Puisque les parenthèses servent aussi à délimiter les expressions, un tuple constitué d'un seul élément doit se déterminer par une virgule.

```
>>> (1)
1
>>> type((1))
<class 'int'>
>>> (1,)
(1,)
>>> type((1,))
<class 'tuple'>
```

Warning : Cette particularité est régulièrement source d'erreurs, et l'on oublie trop aisément la virgule finale.

14.3.3 Chaines de caractères

Une chaine de caractères ("string", abrégé en "str"), comme son nom l'indique, est un ensemble de lettres, que l'on utilisera pour des mots, des phrases, voire des textes entiers. Une chaine est représentée comme une séquence de caractères (chaine de longueur 1), même s'il n'existe aucune différence fondamentale entre ces deux concepts. La fonction de conversion est `str()`.

Une chaine est délimitée par des apostrophes `'` ou par des guillemets `"`.

Note : Contrairement à certains langages qui font la différence entre les apostrophes et les guillemets pour entourer, il n'y en a aucune en Python : les deux délimiteurs sont totalement équivalents.

Il s'agit d'un type immuable car il est impossible de changer individuellement l'un des caractères, mais il est possible d'accéder individuellement aux caractères par leurs indices.

```
>>> s = 'Hello'
>>> type(s)
<class 'str'>
>>> isinstance(s, str)
True
>>> s[1]
'e'
>>> s[1] = 'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> "string"
'string'
```

Warning : Il est impossible d'utiliser le délimiteur choisi dans la chaine elle-même : il est nécessaire de l'échapper à l'aide de l'antislash `\`.

```
>>> 'l'enfant'
  File "<stdin>", line 1
    'l'enfant'
           ^
SyntaxError: invalid syntax
```

Pour éviter cette erreur, on peut soit choisir d'entourer la chaine avec des guillemets, soit faire précéder l'apostrophe intérieur d'un antislash :

```
>>> "l'enfant"
"l'enfant"
>>> 'l\'enfant'
"l'enfant"
```

Warning : Rappelons que Python est un langage au typage fort : il ne considère donc pas `'1'` comme un nombre :

```
>>> '1' + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

14.3.4 Dictionnaires

Un dictionnaire est un conteneur, délimités par des accolades, dont les éléments sont accessibles par une clé. Les paires clé-valeur s'écrivent `clé: valeur` et sont séparés par des virgules. Enfin, il est possible de changer la valeur associée à une clé ou d'ajouter une nouvelle paire par une simple affectation. La fonction intégrée dédiée à la création de dictionnaires est `dict()`.

```
>>> dic = {'a': 'val', 3: 'x', 'key': 124}
>>> type(dic)
<class 'dict'>
>>> dic['a']
'val'
>>> dic[3]
'x'
>>> dic['key'] = 9
>>> dic
{'a': 'val', 3: 'x', 'key': 9}
>>> dic['new'] = 'Hello'
>>> dic
{'a': 'val', 'new': 'Hello', 3: 'x', 'key': 9}
```

Remarquons que la nouvelle valeur se retrouve en deuxième position : les dictionnaires ne sont pas des ensembles ordonnés, c'est à dire que l'on ne peut prédire à quel endroit s'ajoutera une paire. Ils ne sont pas adaptés pour stocker des objets qui doivent restés classés quoiqu'il arrive (pour cela il convient de se tourner vers les listes ou les tuples).

14.3.5 Ensembles

Les ensembles ("set") Python sont très similaires aux ensembles mathématiques : il s'agit d'un ensemble d'objets non ordonnés et uniques. On peut les créer soit en utilisant la fonction `set()` (remarquons qu'il existe la fonction `frozenset()` pour créer un ensemble immuable), soit en utilisant les accolades `{}` et en plaçant à l'intérieur la liste des objets :

```
>>> deck = {3, 4, 12, 7, 9}
>>> deck
{3, 4, 12, 7, 9}
```

Il ne faut pas confondre les dictionnaires et les ensembles, bien que la syntaxe utilise le même délimiteur.

Voici la liste des opérateurs spécifiques aux ensembles (d'autres opérations communes à tous les conteneurs seront présentées bientôt) et de leurs effets :
– `a & b` : intersection de `a` et `b`, soit l'ensemble des éléments communs aux deux ensembles ;
– `a | b` : union de `a` et `b`, c'est à dire tous les éléments de chaque ensemble (en ne comptant qu'une fois ceux en communs) ;
– `a - b` : sous-ensemble des éléments de `a` qui ne sont pas dans `b` (différence) ;
– `a ^ b` : sous-ensembles des éléments de `a` et `b` qui ne sont communs à aucun des deux ensembles.
Ils ne trouvent que rarement une utilité, mais je vais ici donner un exemple, qui permettra de plus de mettre en applications les opérations. Imaginons que je souhaite collectionner les quinze cartes d'un certain jeu, (numérotées de 1 à 15) et que je ne m'intéresse pas à la quantité que je possède (un peu comme si on collait ces cartes dans un album). Dans ce cas les ensembles seront parfaitement adaptés pour représenter ma collection. Finalement je peux agrandir ma collection un achetant un *blister* de trois cartes :

```
>>> cards = set(range(1, 16))
>>> deck < cards
True
>>> cards - deck
{1, 2, 5, 6, 8, 10, 11, 13, 14, 15}
>>> blister = {2, 7, 11}
>>> deck | blister
{2, 3, 4, 7, 9, 11, 12}
>>> deck & blister
{7}
>>> blister - deck
{2, 11}
```

Nous avons d'abord vérifié que nos cartes était bien un sous-ensemble des de la collection, puis calculé successivement : les cartes qui nous manquent, les cartes que l'on a après avoir acheté le blister, les cartes en doublons puis les nouvelles cartes apportées par le blister. Les opérations peuvent bien sûr être enchainées :

```
>>> {1, 3} | {5} | {4, 10, 12}
{1, 3, 4, 5, 10, 12}
```

14.3.6 Autres conteneurs

Le module `collections` propose différents types de conteneurs qui peuvent être utilisés lorsque l'on a des besoins précis. Par exemple, afin de répondre à une question qui se pose souvent, on peut construire des dictionnaires ordonnées à partir de la classe `OrderedDict` du module standard . Toutefois, une réflexion plus ou moins poussée permet de trouver une méthode plus élégante qu'un dictionnaire ordonné (par exemple en choisissant une liste de tuple à la place...). On y trouvera aussi un objet pour fabriquer des compteurs (`Counter`).

Notons enfin l'existence des types `bytes` et `bytearray`, qui permettent de représenter des chaines de caractères ASCII.

La fonction `range()` retourne un objet de ce type, qui représente une suite de nombres entiers.

14.3.7 Opérations

De nombreuses opérations sont définies sur les conteneurs :
– `x in s` : retourne vrai si l'objet `x` se trouve dans le conteneur `s`, sinon faux ;
– `x not in s` : retourne vrai si l'objet `x` ne se trouve pas dans la conteneur `s`, sinon faux ;
– `s1 + s2` (séquences) : concaténation des séquences `s1` et `s2` ;
– `s * n, n * s` (séquences) : création puis concaténation de `n` copies superficielles ("swallow") de `s` (`n` doit être un entier positif).
Nous avons dit que les tuples étaient immuables, or nous présentons ici des opérations entre eux : en fait, crée un nouveau tuple, dans lequel il place les éléments des deux tuples d'origine.

Warning : Les copies réalisées avec la "multiplication" sont superficielles, c'est à dire que seules les références sont copiées :

```
>>> l = [[]]
>>> l *= 3
>>> l[0].append(3)
>>> l
[[3], [3], [3]]
```

De plus, il existe quelques fonctions qui peuvent être utiles.

len(*s*)
Retourne la longueur dans le conteneur `s`.

min(*s*)
Retourne le plus petit élément dans le conteneur `s`.

max(*s*)
Retourne le plus grand élément dans le conteneur `s`.

Note : Si `s` est un dictionnaire, alors toutes les opérations se font au niveau des clés.

Les séquences possèdent les deux méthodes suivantes :
– `s.index(i)` : retourne l'indice de la première occurrence de l'objet `i` dans la séquence `s` ;
– `s.count(i)` : compte le nombre d'occurrences de l'objet `i` dans la séquence `s`.
Mettons un peu en pratique toutes ces opérations :

```
>>> d = {'a': 1, 'b': 5}
>>> 'a' in d
True
>>> 5 in d
False
>>> max(d)
'b'
>>> t = (1, 3) + (5, 6, 1)
>>> t
(1, 3, 5, 6, 1)
>>> 8 not in t
True
>>> len(t)
5
>>> t.count(1)
```

```
2
>>> t.index(3)
1
>>> min(t)
1
>>> [1, 2] * 4
[1, 2, 1, 2, 1, 2, 1, 2]
```

14.3.8 Slicing

Le slicing permet d'extraire une sous-séquence d'une autre séquence (dans le cas où un seul élément est sélectionné, il est simplement retourné). La syntaxe générale est `seq[i:j:k]`, où `i`, `j` et `k` sont des entiers, éventuellement négatifs : la sous-séquence sera composée de tous les éléments de l'indice i jusqu'à l'indice j-1, par pas de k (ce dernier nombre peut être omis).

Si `i` ou `j` sont négatifs, cela est équivalent à l'indice `n-i` et `n-j`, où `n` est la taille de la séquence (on a écrit explicitement le signe moins des variables `i` et `j`). Si `k` est négatif, alors l'ordre des éléments est inversé.

```
>>> l = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> l[-3]
7
>>> l[7]
7
>>> l[2:5]
[2, 3, 4]
>>> l[-6:8]
[4, 5, 6, 7]
>>> l[-6:-2]
[4, 5, 6, 7]
>>> l[5:2:-1]
[5, 4, 3]
>>> l[2:9:3]
[2, 5, 8]
```

14.4 Autres objets

14.4.1 L'objet nul

L'objet nul est `None`. Il correspond à la valeur `NULL` (PHP, C...), ou encore `nil` (Pascal) d'autres langages.

Dans la plupart des cas, il signifie l'absence de valeur. Il est par exemple utilisé comme valeur de retour par les fonctions qui ne retournent aucune valeur explicitement.

14.4.2 Type

`type()`, déjà utilisée de nombreuses fois, renvoie un objet `type`. Il s'agit donc du constructeur d'une classe.

```
>>> type
<class 'type'>
>>> type(str)
<class 'type'>
```

14.5 Résumé

Dans ce chapitre, nous avons étudié de manière superficielle les principaux types du langage Python, ainsi que leurs opérations :
– les nombres : entiers, à virgule, complexes et booléens ;
– les conteneurs et les séquences : listes, tuples, chaines de caractères, dictionnaires ;
– les objets nul et type.
Les fonctions `isinstance()` et `type()` permettant d'étudier le type d'un objet ont aussi été abordées.

14.6 Exercices

1. Essayez les différentes opérations sur les divers types de nombres. Observez les résultats lorsque les types étaient différents à la base.
2. De même, créez diverses listes et entrainez-vous à les manipuler.
3. Choisir un nombre (par exemple `1`) et divisez-le mille fois par `10`. Multipliez-le ensuite mille fois par `10` (on pourra attendre d'avoir lu le chapitre sur les boucles). Expliquez le résultat.
4. Créer un système de permissions qui fonctionne comme suit :
   – chaque permission est associée à un nombre (e.g. 1 : "peut écrire un article", 2 : "peut éditer un article"...) ;
   – chaque groupe se voit attribué plusieurs permissions ;
   – un utilisateur peut appartenir à plusieurs groupes ;
   – tester si un utilisateur a certaines permissions (on pourra attendre la lecture du prochain chapitre pour répondre à cette question, voire celle sur les fonctions afin de simplifier le code).

CHAPITRE 15

Structures de contrôle

15.1 Construction de prédicats

Un prédicat est une expression qui possède une valeur de vérité : par exemple, “La variable x est un nombre plus grand que deux” est un prédicat, qui pourra être vrai ou faux selon la valeur de la variable, mais, dans tous les cas, le résultat sera sans ambigüité. Par contre, la phrase “La variable x est grande” n’est pas un prédicat, car il n’y a pas de comparaison.

15.1.1 Valeur de vérité des objets

Tout objet en Python possède naturellement une valeur de vérité. En fait, tous les objets sont équivalents à “vrai”, sauf les suivants :
– objets particuliers : `False`, `None` ;
– les conteneurs vides : `()`, `[]`, `{}` (dictionnaire), `''`, etc. ;
– les nombres qui sont zéros : `0`, `0.0`, `0j`, etc.
Afin de simplifier les termes, on dira simplement qu’un objet est vrai/faux, même si ce n’est qu’une équivalence. On peut vérifier cette particularité avec la fonction `bool()` :

```
>>> bool([])
False
>>> bool([1, 2])
True
>>> 1 is True
False
>>> 1 == True
True
```

15.1.2 Combinaison d’expressions

L’opérateur `not` retourne `True` si l’expression qui suit est fausse, et inversement, elle retourne `False` si ce qui vient après est vrai :

```
>>> not True
False
>>> not False
True
```

Finalement, les deux mots-clés `and` et `or` permettent de combiner plusieurs expressions :
– `x and y` : l'expression `x` est d'abord évaluée et sa valeur est retournée si elle est fausse. Sinon, `y` est évaluée et sa valeur est retournée.
– `x or y` : l'expression `x` est d'abord évaluée et sa valeur est retournée si elle est vraie. Sinon, `y` est évaluée et sa valeur est retournée.
Ces relations sont résumées dans les deux tableaux suivants (le second traduit le premier en termes de valeurs de vérité).

A	B	A and B	A or B
False	False	A	B
True	False	B	B
False	True	A	A
True	True	B	A

A	B	A and B	A or B
False	False	False	False
True	False	False	True
False	True	False	True
True	True	True	True

Il est important de noter que la valeur retournée n'est *pas* convertie en booléen. Cette construction est utile par exemple si on veut attribuer une valeur par défaut à un objet vide :

```
>>> True and False
False
>>> s = '' or 'défaut'
>>> s
'défaut'
>>> 1 or 2
1
>>> 1 and 2
2
```

15.1.3 Comparaisons

Un opérateur permet de comparer deux objets, selon différents critères. Voici la liste de ces opérateurs, ainsi que leur signification :
– `a is b` : vrai si `a` est identique à `b` ;
– `a is not b` : vrai si `a` n'est pas identique à `b` ;
– `a == b` : vrai si `a` est égal à `b` ;
– `a != b` : vrai si `a` est différent de `b` ;
– `a > b` : vrai si `a` est plus grand que `b` ;
– `a < b` : vrai si `a` est plus petit que `b` ;
– `a >= b` : vrai si `a` est plus grand ou égal à `b` ;
– `a <= b` : vrai si `a` est plus petit ou égal à `b`.

Note : En ce sens, les deux mots-clés `in` et `not in` peuvent être vus comme des opérateurs de comparaison.

Le symbole d'égalité == dénote une égalité superficielle, dans le sens où deux objets peuvent être égaux mais pas identiques :

```
>>> 1.0 is 1
False
>>> 1.0 == 1
True
```

La même distinction existe entre `is not` et `!=`.

Les opérateurs ont tous le même ordre de priorité, qui est plus faible que celle des opérateurs arithmétiques (+, −...), et ils seront donc lus de la gauche vers la droite : il est permis d'écrire `a op1 b op2 c` (où `a`, `b`, et `c` sont des expressions, `op1` et `op2` sont des opérateurs), qui sera interprété comme ``a op1 b and b op2 c`:

```
>>> a = 3
>>> 1 < a < 10
True
>>> 1 < a > 2
True
```

La dernière ligne peut paraitre étrange, mais elle est tout à fait valable : en effet, elle est interprétée comme `1 < a and a > 2`.

15.1.4 Conclusion

Les différents opérateurs de combinaison d'expressions permettent de construire des prédicats complexes à partir de prédicats plus simples, comme des comparaisons ou des opérations arithmétiques. Ces prédicats jouent réellement le rôle de booléen et peuvent être utilisés partout là où un booléen est attendu.

15.2 Condition — if

Une condition `if` ("si" en anglais) permet d'interpréter ou non une suite d'instructions en fonction de la valeur de vérité d'une expression [1].

```
>>> if False:
...     print(1)
>>> if True:
...     print(2)
2
```

Il est possible de construire des conditions plus évoluées grâce aux mots-clés `elif` (contraction de "else if" : "sinon si") et `else` ("sinon"). Les conditions `elif` sont évaluées tour à tour jusqu'à ce que l'une soit vérifiée. Si aucune ne l'est, alors les instructions du bloc `else` sont évaluées :

```
>>> a = 3
>>> if a > 4:
...     print('"a" is higher than 4')
... elif a > 2:
...     print('"a" is higher than 2 and lower than or equal to 4')
... elif a > 0:
...     print('"a" is higher than 0 and lower than or equal to 2')
... elif a == 0:
...     print('"a" is equal to 0')
... else:
...     print('"a" is lower than 0')
"a" is higher than 2 and lower than or equal to 4
```

N'hésitez pas à changer la valeur de `a` et à tester à nouveau.

1. En vertu de ce qui a été dit dans la conclusion de la section précédente, le `True` et le `False` peuvent être remplacés par n'importe quel prédicat.

15.3 Boucle conditionnelle — while

Une boucle conditionnelle `while` ("tant que") exécute une suite d'instructions tant qu'une condition vaut vraie.

```
>>> a = 0
>>> while a < 5:
...     print(a)
...     a += 1
0
1
2
3
4
```

Warning : Dans le cas où vous auriez indiqué un prédicat tout le temps vrai — on parle de boucle infinie —, il est possible d'interrompre la boucle en appuyant sur `Ctrl+C`. S'il s'agit d'une interface graphique, alors il est nécessaire de demander au gestionnaire des programmes de tuer le processus en question.

Toutefois, il existe des situations les boucles infinies peuvent s'avérer utile, à condition qu'elle puise être interrompue quand leur rôle est terminé. Nous y reviendrons à la fin du chapitre car il nous manque encore un outil important : le mot-clé `break`.

15.4 Itération — for

La boucle `for` est construite sur le modèle `for obj in cont` ("pour ... dans ..." en anglais). Elle consiste à sélectionner un à un les objets du conteneur `cont`, en les plaçant successivement dans la variable `obj`. Ainsi, à chaque itération, cette variable prend une valeur différente, qui est celle d'un objet du conteneur.

```
>>> for i in ['a', 'b', 'c']:
...   print(i)
a
b
c
```

Warning : Dans le cas particulier d'une séquence, les objets sont parcourus dans l'ordre, mais il n'est pas possible de prédire l'ordre de parcours pour un conteneur général, comme un dictionnaire, car aucun ordre n'est défini.

Il est fortement déconseillé de modifier les objets sur lesquels on itère dans une boucle `for`, car le risque est grand que l'interpréteur ne fasse pas ce que l'on attendait de lui, ou même qu'il lève une exception `RuntimeError` :

```
>>> dic = {'a': 1, 'b': 2}
>>> for i in dic:
...     del dic[i]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: dictionary changed size during iteration
```

Ici, l'interpréteur est mécontent car la taille du dictionnaire a changé au cours de la boucle, et il ne sait plus où il en est.

Il est nécessaire de faire une petite digression afin de présenter l'inestimable fonction `range(ini, fin)()`, qui crée un objet représentant les entiers de `ini` à `fin-1` (la structure détaillée est donnée dans la section *Fonctions built-in*) :

```
>>> range(3, 10)
range(3, 10)
>>> list(_)
[3, 4, 5, 6, 7, 8, 9]
```

Combinée avec la structure `for ... in`, elle remplace avantageusement la boucle `for` (à la syntaxe parfois compliquée) des autres langages, qui permet seulement d'énumérer une liste de nombres :

```
>>> for i in range(5):
...     print(i)
0
1
2
3
4
```

15.5 Autres techniques pour les boucles

Les mots-clés `continue` et `break` permettent d'influencer le déroulement des boucles `for` et `while`, et un bloc `else` peut leurs être ajoutées.

15.5.1 Continue

Le mot-clé `continue` force l'interpréteur à revenir au début de la boucle et à entamer l'itération suivante, sans exécuter les instructions qui suivent ce mot-clé.

```
>>> for i in range(4):
...     if i == 2:
...         continue
...     print(i)
0
1
3
```

Dans cet exemple, on voit que lorsque la variable `i` vaut 2, la condition devient vraie et l'instruction `print(i)` est ignorée.

15.5.2 Break

Le mot-clé `break` permet d'interrompre l'exécution d'une boucle, quel que soit son avancement.

```
>>> for i in range(20):
...     if i == 4:
...         break
...     print(i)
0
1
2
3
```

Nous voyons que dès que `i` prend la valeur 4, l'instruction break termine l'exécution de la boucle même si l'affichage des nombres aurait dû se poursuivre jusqu'à 19.

15.5.3 Else

Il est possible d'ajouter une clause `else` finale à une boucle. Le bloc d'instructions ne sera exécuté que si la boucle s'est terminé normalement, c'est à dire qu'aucune instruction `break` n'a été rencontrée [2].

Voici un premier cas où la boucle se termine et où la clause `else` est exécutée :

```
>>> for i in range(6):
...     print(i)
... else:
...     print("end")
0
1
2
3
4
5
end
```

Maintenant, regardons un exemple où la boucle est arrêtée :

```
>>> for i in range(6):
...     if i == 4:
...         break
...     print(i)
... else:
...     print("end")
0
1
2
3
```

15.6 Instruction `pass`

`pass` ne fait strictement rien. Il est utilisé lorsque l'interpréteur attend un bloc de code mais que l'on ne souhaite rien écrire pour un temps, ou ne rien faire. Nous verrons son utilisation pratique dans le cadre des *Erreurs et exceptions*, des *Fonctions* et des *Classes*.

15.7 Imbrications

Rien n'empêche d'imbriquer plusieurs structures de contrôle (ou blocs) dans les autres, tant que l'indentation est respectée. Le code suivant affiche la liste des nombres pairs ou impairs compris entre 0 et 10 (exclus) selon que la variable `n` soit pair ou impair :

```
>>> n = 12
>>> for i in range(1, 10):
...     if (n % 2) == 0 and (i % 2) == 0:
...         print(i)
...     elif (n % 2) != 0 and (i % 2) != 0:
...         print(i)
2
4
```

2. Elles permettent donc, entre autres, de faire ce que font les boucles `foreach` de langages tel que le *PHP*.

```
6
8
```

Les parenthèses autour des opérations de modulo auraient pu être omises, comme l'opérateur de modulo évalué avant l'opérateur d'égalité. L'opérateur `and` est évalué en dernier.

En principe on évitera d'imbriquer plus de trois ou quatre afin de garder une certaine lisibilité du code, d'autant plus que, dans ces cas-là, il existe des manières plus élégantes de disposer le code.

Avant de conclure ce chapitre, donnons un exemple typique de boucle infini : l'affichage d'un menu dans un programme en ligne de commande, qui propose différentes actions à l'utilisateur. Ce dernier choisit une action à effectuer en appuyant sur une touche, et, une fois l'action terminée, il est ramener au menu. Et ainsi de suite jusqu'à ce qu'il appuie sur une touche permettant de quitter le programme (nous omettons les indications du prompt) :

```
while True:
    print('Menu')
    print('Action A : touche A')
    print('...')
    print('Action F : touche F')
    print('Quitter : touche Q')
    choice = input('Quel est votre choix ? ')
    print()

    if choice == 'A':
        print("Exécuter l'action A")
    if choice == 'F':
        print("Exécuter l'action F")
    if choice == 'Q':
        print("Fin du programme")
        break
    print()
```

Notez la présence des guillemets au lieu de l'apostrophe, ce qui permet d'écrire « l'action » sans devoir échapper l'apostrophe après le "l". La présence des `print()` sans argument permet de sauter une ligne.

15.8 Exceptions

Les exceptions sont une sorte de structures de contrôle, mais un chapitre entier leur est réservé du fait de leur spécificité. Ainsi, pour plus de détails, se reporter au chapitre *Erreurs et exceptions*.

15.9 Résumé

Dans ce chapitre nous avons vu comment créer des prédicats et nous en servir pour construire des blocs conditionnels et des boucles, que l'on a ensuite imbriqués.

15.10 Exercices

1. Écrire une boucle qui calcule la somme des dix premiers carrés (c'est à dire $1^2 + 2^2 + \ldots + 9^2$).
2. Affichez à l'écran les quinze premiers multiples d'un nombre que vous choisirez.
3. Écrire une boucle qui affiche tous les nombres divisibles par 3 et par 5 dans un intervalle que vous choisirez.
4. Calculer la longueur d'une chaine de caractères (sans utiliser la fonction `len()`).

5. Écrire une boucle qui permet de compter le nombre de voyelles dans une chaine de caractères.
6. Créer un programme qui doit faire deviner au joueur un nombre entre 1 et 100 tiré aléatoirement (on pensera à utiliser le module `random`). Après chaque entrée, on indiquera au joueur si le nombre est plus grand ou petit.
   On pourra envisager d'ajouter un compteur de tentatives.
7. Écrire un algorithme semblable au précédent, en inversant les rôles : le joueur choisit un nombre *entier* dans un certain intervalle, et l'ordinateur doit deviner ce dernier.
8. Améliorer le programme précédent pour faire deviner un nombre à virgule. L'ordinateur pourra-t-il deviner précisément le nombre ? Penser à adapter l'algorithme.

CHAPITRE 16

Fonctions

16.1 Généralités

16.1.1 Définition

Une fonction se définit à l'aide du mot-clé `def` suivi d'un identifiant puis de parenthèses. Ces dernières contiennent la définition des différents arguments que la fonction peut recevoir : rappelons qu'un argument permet de modifier le comportement d'une fonction en lui fournissant des données supplémentaires, qui permettent ainsi un comportement différent en fonction du contexte. Enfin, la ligne contenant l'instruction `def` se termine par deux points et est suivie d'un bloc d'instructions(indentées). On aura donc la forme suivante :

```
def name(args):
    instructions
```

Une fonction peut retourner une valeur grâce au mot-clé `return` (dans certains langages, on appelle procédure une fonction ne retournant rien, mais la distinction n'est pas importante ici). Il faut noter que le mot-clé return interrompt l'exécution de la fonction, c'est à dire que toute instruction située après sera ignorée (sauf dans un cas que nous verrons dans le chapitre *Erreurs et exceptions*).

Note : En réalité, une fonction retournera toujours une valeur, qui sera `None` si aucun `return` n'est présent.

Voici quelques exemples de définitions de fonctions :

```
>>> def func():
...     return 1
>>> func()
1

>>> def square(n):
...     m = n**2
...     return m
>>> square(3)
9
```

```
>>> def nothing():
...     pass
>>> print(nothing())
None
```

La première fonction n'est pas très utile puisqu'elle est équivalente à entrer directement `1`. Il aurait été possible de simplifier la deuxième fonction en écrivant directement `return n**2`.

16.1.2 Valeurs de retour

Il est possible de retourner n'importe quel type d'objet grâce à `return`.

```
>>> def return_obj(obj):
...     return obj
>>> type(return_obj(1))
<class 'int'>
>>> type(return_obj('string'))
<class 'str'>
>>> type(return_obj([1, 2, 3]))
<class 'list'>
>>> return_obj([1, 2, 3])
[1, 2, 3]
```

S'il n'est pas possible d'utiliser plusieurs `return` pour retourner plusieurs valeurs, il est possible de retourner plusieurs objets par un même `return`, soit explicitement avec un objet de notre choix (tuple, liste, dictionnaire), soit implicitement (ce sera alors un tuple).

```
>>> def tds(n):
...     return n*3, n/2, n**2
>>> tds(4)
(12, 2, 16)
```

16.1.3 Docstrings

Il convient d'associer à chaque fonction une docstring, c'est à dire une chaine de documentation. Celle-ci permet de donner diverses informations sur la fonction. Les docstrings peuvent être utilisées par certains générateurs de documentation pour créer automatiquement la documentation d'un programme.

Une docstring doit respecter quelques conventions :
- la première ligne consiste en une courte description de la fonction (moins d'une ligne), et elle doit débuter par une majuscule et finir par un point ;
- la deuxième ligne doit être vide ;
- les lignes suivantes décrivent d'une manière plus complète la fonction (comment l'appeler, les objets retournés, etc), éventuellement en utilisant des marquages pour les générateurs de documentations.

Seule la première ligne n'est pas optionnelle. Il est possible d'accéder au docstring d'une fonction grâce à l'attribut `__doc__`.

```
def func():
    ''' Do nothing.

    This function does not make anything.
    '''

    pass
```

L'on peut alors afficher la documentation avec `print(func.__doc__)`.

16.1.4 Remarques

Rappelons que les variables dans une fonction sont locales (et donc supprimées par le ramasse-miettes de Python dès que l'interpréteur sort de la fonction).

```
>>> def func():
...     a = 1
>>> func()
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

Une fonction est un objet appellable. Il est donc nécessaire d'utiliser des parenthèses même si elle ne prend aucun argument. Dans le cas contraire, l'on aurait simplement accès à l'objet fonction lui-même, et le code à l'intérieur ne serait pas exécuté.

```
>>> def re1():
...     return 1
>>> re1
<function re1 at ...>
>>> re1()
1
```

Enfin, il est fortement déconseillé d'afficher du texte (via `print()` par exemple) directement dans le corps d'une fonction (du moins, pour une autre raison que du débuggage), car cela peut conduire à des effets indésirables.

16.2 Arguments

16.2.1 Général

Il est possible de créer une fonction avec autant d'argument que l'on souhaite. Dans l'exemple qui suit, nous utilisons trois arguments, mais l'on aurait très bien pu en utiliser deux comme cinq ou vingt.

```
>>> def add(a, b, c):
...     return a + b + c
>>> add(1, 2, 3)
6
>>> add(2, -4, 7)
5
```

Il est toutefois important de signaler que, lorsque l'on appelle la fonction, les arguments doivent être dans le même ordre qu'ils le sont dans la définition.

```
>>> def test(string, integer):
...     print('The string is '+ string +' and the integer is '+ str(integer))
>>> test('abc', 1)
The string is abc and the integer is 1
>>> test(1, 'abc')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in test
TypeError: Can't convert 'int' object to str implicitly
```

Ici, une exception est levée car, en théorie, il faut fournir en premier une chaine puis un nombre. Or l'on fait l'inverse, ce pour quoi le programme n'a pas été prévu à la base.

16.2.2 Arguments par défaut

Dans la définition de la fonction, il est possible de donner une valeur par défaut à certains arguments, de sorte que l'on ait pas à fournir tous les arguments lorsque l'on appelle la fonction. La valeur par défaut ne sera utilisée que si aucune valeur n'est fournie pour l'argument en question. Une valeur par défaut se définit par `argument=valeur`.

Warning : Il est nécessaire de placer les arguments ayant une valeur par défaut en dernier, sans quoi une exception `SyntaxError` sera levée.

```
>>> def add(n, m=1):
...     return n + m
>>> add(2, 5)
7
>>> add(2)
3
```

L'on peut utiliser une variable pour indiquer la valeur par défaut d'un argument. Toutefois la valeur de la variable utilisée est celle qu'elle avait au moment de la définition de la fonction.

```
>>> i = 5
>>> def f(arg=i):
...     return arg
>>> i = 6
>>> f()
5
```

Warning : Il est fortement déconseillé d'utiliser un objet mutable comme valeur par défaut, car, comme nous l'avons vu au-dessus, la valeur par défaut n'est évaluée qu'une seule fois.

```
>>> def f(a, L=[]):
...     L.append(a)
...     return L
>>> for i in range(3):
...     print(f(i))
[0]
[0, 1]
[0, 1, 2]
```

Note : Il est fortement conseillé de toujours mettre une valeur par défaut à tous les arguments d'une fonction, *a fortiori* lorsqu'ils ne sont pas forcément nécessaires.

16.2.3 Arguments nommés

Lors de l'appel d'une fonction il est possible d'attribuer explicitement la valeur d'un argument. Cela se fait en écrivant `argument=valeur`.

Il faut aussi faire attention à ne pas fournir deux (ou plus) valeurs pour un même argument.

```
>>> def test(string, integer=2):
...     print('string is '+ string +' and integer is '+ str(integer))
>>> test('abc', 1)
string is abc and integer is 1
>>> test('abc', integer=1)
```

```
string is abc and integer is 1
>>> test(string='abc', integer=1)
string is abc and integer is 1
>>> test(integer=1, string='abc')
string is abc and integer is 1
>>> test(string='abc')
string is abc and integer is 2
```

Toutefois, au même titre que l'attribution d'une valeur par défaut, il est nécessaire de placer les arguments nommés à la fin.

16.2.4 Capture d'arguments non définis

Il existe deux sortes d'arguments spéciaux qui permettent de "capturer" les arguments fournis par l'utilisateur mais qui ne sont pas définis dans la fonction :
– `**kwargs`, qui sera un dictionnaire des arguments nommés (où l'argument sera la clé) ;
– `*args`, qui sera un tuple des autres arguments.
Cela permet donc d'obtenir une liste d'arguments de taille variable, et non définie à l'avance, ce qui peut être un gage de souplesse. `*args` doit absolument être placé avant `**kwargs`.

```
>>> def func(n, *num, **kwargs):
...     return(n, num, kwargs)
>>> func(1, 2, 3, 4, a=5, b=6)
(1, (2, 3, 4), {'a': 5, 'b': 6})
```

Il est bien sûr possible de manipuler les variables dans le corps de la fonction comme un tuple et un dictionnaire normaux (puisque c'est ce qu'ils sont). Toutefois il faut prendre garde à ne pas utiliser les astériques en dehors de la définition.

Note : Les noms `kwargs` et `args` sont utilisés par conventions, mais ils peuvent être nommés n'importe comment. Ce qui compte est le nombre d'astérisques devant le nom.

Il est aussi possible d'utiliser `**kwargs` et `*args` pour transmettre des arguments à une fonction. Le terme en anglais pour cette opération est *unpack* (dépaquetter).

```
>>> def add(a, b, c):
...     return a + b + c
>>> t = (1, 2, 3)
>>> add(*t)
6
```

Sans l'astérisque, nous aurions eu une exception `TypeError`. Il est possible de faire de même pour transmettre des paramètres nommés.

16.3 Fonctions récursives

Une fonction récursive est une fonction qui s'appelle elle-même. Définissons par exemple une fonction calculant la factorielle d'un nombre.

```
>>> def fact(n):
...     if n == 0:
...         return 1
...     else:
```

```
...         return n * fact(n-1)
>>> fact(4)
24
```

La récursivité sera abordée en détails plus loin.

16.4 Fonctions anonymes

Il s'agit d'une fonctionnalité héritée de langages de programmation fonctionnelle. Le mot-clé `lambda` permet de créer une fonction anonyme, c'est à dire une fonction employée directement là où l'on en a besoin, au lieu de la définir auparavant. Une expression lambda peut donc être utilisée n'importe où un objet fonction devrait être requis en temps normal.

```
>>> (lambda a: a**2)(2)
4
>>> f = lambda a: a**2
>>> f(2)
4
>>> def power(n):
...     return lambda m: m**n
>>> f = power(3)
>>> f(2)
8
>>> power(3)(2)
8
```

Il est bien entendu possible de définir une expression lambda utilisant plusieurs arguments.

```
>>> f = lambda a, b: a + b
>>> f(1, 2)
3
```

Remarquons qu'une seule expression ne peut être définie dans une fonction lambda, et que le `return` est implicite (le rajouter lève une exception `SyntaxError`).

Note : Les fonctions lambda peuvent compliquer la lecture du code, et il est conseillé pour cette raison de ne pas en utiliser trop. De même, dès qu'une même fonction lambda est utilisée à plusieurs endroits, il faut songer sérieusement à définir une fonction normale.

16.5 Generators

Un générateur est une fonction qui, au lieu de retourner des objets via `return`, utilise `yield`. D'une certaine manière, l'on peut voir le processus ainsi :
– on appelle la fonction, qui retourne un objet générateur ;
– le premier appelle de `next()` lance l'exécution de la fonction ;
– au premier `yield` rencontré, l'exécution s'arrête et un objet est retourné ;
– on appelle à nouveau `next()`, et la fonction reprend son exécution là où elle avait été stoppée par `yield` ;
– l'exécution s'arrête à nouveau dès qu'un yield est rencontré ;
– et ainsi de suite.
Pour commencer, prenons un exemple de générateur simple :

```
>>> def gen():
...     yield 1
...     yield 2
...     yield 3
>>> g = gen()
>>> g
<generator object gen at ...>
>>> next(g)
1
>>> next(g)
2
>>> next(g)
3
>>> next(g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

L'on observe qu'une exception `StopIteration` est levée dès que le générateur est "épuisé". Ceci permet de contrôler ce qui se passe dès qu'il ne peut plus retourner d'objet.

Les générateurs peuvent se comporter comme des itérateurs, c'est à dire qu'une boucle `for` permet d'itérer sur les objets qu'il contient. À ce moment-là, la boucle `for` appelle implicitement la fonction `next()` à chaque itération, et elle se termine dès que l'exception `StopIteration` est levée.

Comme exemple nous allons définir la suite de Fibonacci.

```
>>> def fibo(n):
...     a, b = 0, 1
...     while b < n:
...         yield b
...         a, b = b, a + b
>>> for i in fibo(6): print(i)
1
1
2
3
5
>>> g = fibo(3)
>>> next(g)
1
>>> sum(fibo(6))
12
```

Le calcul de la somme s'explique du fait que la fonction `sum()` accepte un itérateur.

16.6 Fonctions built-in

Quelques fonctions n'appartenant à aucun module sont définies sans qu'il ne soit nécessaire des les importer (se référer au chapitre *Modules* pour plus d'informations). On les appelle fonctions *built-in* (que l'on pourrait traduire par "intégrée", mais le terme anglais restera utilisé). Elles son généralement souvent utilisées, d'où leur disponibilité immédiate.

Ici sera dressée une liste des fonctions les plus importantes, la découverte des autres étant laissée en exercice.

dir([*object*])
: Retourne une liste des toutes les méthodes et attributs de `object` s'il est fourni, sinon retourne une liste de

tous les objects du contexte courant. Cette fonction est essentiellement utilisée dans l'interpréteur interactif pour tester les objets.

eval(*expression*)
Cette fonction analyse et exécute `expression` (qui doit être une chaine).

```
>>> x = 1
>>> eval('x + 2')
3
```

globals()
Retourne un dictionnaire représentant les variables présente dans le contexte global.

hasttr(*object*, *attribut*)
Retourne `True` si l'objet possède l'attribut en question.

help([*object*])
Affiche l'aide au sujet d'un objet si `object` est fourni, sinon affiche l'aide générale. Cette fonction est destinée à être utilisée dans l'interpréteur interactif.

input([*prompt*])
Si le paramètre `prompt` est fourni, alors il sera affiché sur la sortie standard. Dans tous les cas, la fonction lit une ligne écrite (dont saisie est interrompue par la touche `entrée`) et la retourne sous forme de chaine.

```
>>> name = input('Enter your name: ')
Enter your name: Harold
>>> print(name)
Harold
```

isinstance(*object*, *class*)
Retourne `True` si `object` est une instance de `class`. `class` peut aussi être un tuple contenant plusieurs classes, et la fonction retournera `True` si `object` est une instance de l'un d'entre eux. Il s'agit de LA manière la plus correcte de vérifier le type d'un objet.

```
>>> if isinstance(1, int):
...     print('1 is an instance of int.')
1 is an instance of int.
>>> if isinstance(1, (int, str)):
...     print('1 is an instance of int or str.')
1 is an instance of int or str.
>>> if isinstance("1", (int, str)):
...     print('"1" is an instance of int or str.')
"1" is an instance of int or str.
```

len(*seq*)
Retourne la longueur de la séquence `seq`.

locals()
Fait de même que `globals()`, mais avec le contexte local.

print(*[obj1, ..., objn][, sep=' '][, end='n']*)
Affiche `obj1` jusque `objn`, séparés par `sep`, en ajoutant `end` à la fin. `sep` et `end` doivent être des chaines.

range([*start=0*], *stop*[, *step=1*])
Retourne un objet `range`, qui est un objet itérable équivalent à une liste d'entiers, commençant à `start` et finissant à `end - 1`, et séparés par le pas `sep`.

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

```
[2, 3, 4]
>>> list(range(1, 9, 2))
[1, 3, 5, 7]
>>> list(range(0, -3, -1))
[0, -1, -2]
```

repr(*obj*)
Affiche la représentation (définie par la méthode `__repr__()` de `obj`.

reversed(*seq*)
Retourne l'inverse de la séquence passée en argument. Ceci est équivalent à `seq.reverse()`, exceptée que cette fonction retourne un nouvel objet, et peut donc être utilisée dans une boucle `for`.

```
>>> for i in reversed(range(3)):
...     print(i)
2
1
0
```

round(*x*[, *n=0*])
Arrondit `x` à `n` décimales.

sorted(*seq*)
Retourne une séquence triée à partir de `seq`. Cette fonction est équivalent à appeler la méthode `seq.sort()`, excepté le fait qu'elle retourne un nouvel objet (comme `reversed()`).

sum(*iterable*)
Permet de sommer un itérable contenant uniquement des nombres (de n'importe quels types, qui peuvent être mélangés).

Quelques exemples d'utilisations :

```
>>> sum([1+4j, 3.3, -2])
(2.3+4j)
```

16.7 Exercices

1. Écrire une fonction qui permet de convertir des francs en euros.
   Pour rappel : 1€ = 6.55957F.
2. Écrire une fonction qui calcule le périmètre d'un cercle de rayon R (qui sera donc l'argument).
   Pour rappel : le périmètre est P = 2×π×R. On pourra prendre π = 3.14 ou, mieux, chercher un module qui contient ce nombre et l'utiliser.
3. Écrire une fonction qui prend en argument un nombre n et qui calcule n ! (n factorielle), sans utiliser la récursivité.
   Rappel : n ! correspond à la multiplication des n premiers entiers avec la convention que 0 ! = 1. Ainsi, par exemple, 4 ! = 1×2×3×4 = 24.
4. Écrire une fonction qui prend deux arguments, l'un d'eux étant une fonction. La première fonction doit appliquer la fonction passée en argument au second argument, et renvoyer le résultat.
5. Écrire une fonction qui prend une date de naissance en argument et qui retourne le nombre de jours écoulés depuis.
   On pensera aux années bissextiles ; une année est bissextile si :
   – elle est divible par 4 mais par par 100 ;
   – elle est divisible par 400.
   On pourra utiliser le module datetime pour obtenir la date du jour courant automatiquement.

6. Écrire une fonction similaire à `range()`, mais qui accepte des flottants en argument.

CHAPITRE 17

Classes

17.1 Introduction

17.1.1 Définition

Un objet est construit à partir d'une classe. On dit que l'on crée une instance d'une classe. Les classes peuvent être vues comme une description de l'objet, qui rassemble ses propriétés (les attributs) et comment agir sur ces dernières (les méthodes).

Une classe est déclarée à l'aide du mot-clé `class`, suivi d'un nom puis de deux points. Suit ensuite un bloc d'instructions. De même que pour les fonctions, il est possible de définir une docstring.

```
class Name:
    '''Docstring.'''
    instructions
```

Note : Par convention, les noms des classes sont en `CamelCase`.

Une classe est un objet callable : ainsi, pour créer un objet à partir d'une classe, il faut écrire son nom suivi de parenthèses.

```
>>> class Vector:
...     pass
>>> v = Vector()
>>> v
<__main__.Vector object at ...>
```

Note : Il est possible de mettre des parenthèses après le nom de la classe, mais ceci est tout à fait optionnel : `class Name():` est tout aussi correct que `class Name:`.

17.1.2 Attributs

Il est possible d'accéder aux attributs d'un objet grâce au point. Il est aussi possible de définir un attribut par simple assignation.

```
>>> class Vector:
...     x = 3
...     y = 4
>>> v = Vector()
>>> v.x, v.y
(3, 4)
>>> v.z = 6
>>> v.z
6
>>> v.z + 3
9
>>> del v.z
>>> v.x = v.x + 4
>>> v.x
7
```

Note : Bien qu'il soit possible de créer ou modifier directement un attribut en Python, il est plutôt déconseillé de le faire : mieux vaut passer par des méthodes dédiées à cet effet.

Il est possible d'accéder récursivement aux attributs d'un objet en utilisant plusieurs points.

```
>>> class CVector:
...     pass
>>> cv = CVector
>>> cv.affix = 1+3j
>>> cv.affix
(1+3j)
>>> cv.affix.imag
3.0
```

Les attributs d'une classe sont encapsulés dans celle-ci et ne sont donc pas visible depuis l'extérieur.

```
>>> v.x
7
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

17.1.3 Méthodes

Une méthode consiste simplement en une fonction définie à l'intérieur de la classe et dont le premier paramètre est **toujours** une référence à l'instance (par convention, l'on utilise `self` — *soi-même* en anglais).

Ainsi `self` désigne, dans le corps de la méthode, l'instance qui contient la méthode.

```
>>> class Vector:
...     x = 3
...     y = 4
...     def norm(self):
...         return (self.x**2 + self.y**2)**(1/2)
...     def homothety(self, n):
...         self.x = self.x * n
...         self.y = self.y * n
>>> v = Vector()
>>> v.norm()
```

```
5.0
>>> v.homothety(3)
>>> v.x, v.y
(9, 12)
```

17.1.4 Constructeur

Le constructeur est une méthode spéciale (il en existe d'autres, que nous verrons par la suite), appelée `__init__()`, qui est appelée lors de l'instanciation de l'objet : cela permet d'effectuer diverses opérations sur chaque nouvel objet à leur création. Par exemple, si l'on reprend le cas précédent, il n'est pas très intéressant d'avoir un vecteur dont les attributs `x` et `y` sont déjà définis. Nous pouvons y remédier ainsi :

```
>>> class Vector:
...     def __init__(self, x=0, y=0):
...         self.x = x
...         self.y = y
...     def norm(self):
...         return (self.x**2 + self.y**2)**(1/2)
>>> v1 = Vector(3, 4)
>>> v1.x, v1.y
(3, 4)
>>> v2 = Vector(-2, 7)
>>> v2.x, v2.y
(-2, 7)
```

17.1.5 Éléments publics et privés

Contrairement à d'autres langages, les concepts de membres privés ou protégés n'existent pas réellement en Python : certaines conventions indiquent qu'un élément est privé, mais l'utilisateur peut toujours choisir d'y accéder. Ainsi, Python part du principe que l'utilisateur n'accédera pas n'importe comment aux éléments.

Un membre protégé commence par un simple underscore, tandis qu'un membre privé commence par deux underscores.

17.2 Méthodes spécifiques

17.2.1 Destructeur

La méthode destructeur, nommée `__del__()` est appelée lorsque l'on détruit l'objet avec `del`.

17.2.2 Opérateurs

Il est possible de surcharger les opérateurs pour une classe, ce qui permet de définir le comportement des objets lorsqu'on les manipule avec des opérateurs. Lles méthodes associées aux opérateurs sont : `__add__()` pour l'addition, `__sub__()` pour la soustraction, `__mul__()` pour la multiplication, `__truediv__()` pour la division, `__floordiv__()` pour la division entière, `__mod__()` pour le modulo, `__pow__()` pour la puissance.

```
>>> class Vector:
...     def __init__(self, x, y):
...         self.x, self.y = x, y
...     def __add__(self, other):
...         if isinstance(other, Vector):
```

```
...             return Vector(self.x + other.x, self.y + other.y)
>>> v1 = Vector(2, 3)
>>> v2 = Vector(-4, 7)
>>> v3 = v1 + v2
>>> v3.x, v3.y
(-2, 10)
```

17.3 Héritage

17.3.1 Introduction

Il est aisé de dériver une classe : pour ce faire, il suffit d'indiquer la ou les classes parentes (séparées par des virgules) entre parenthèses après le nom de la classe enfant.

```
class Name(ParentName1, ..., ParentNamen):
    instructions
```

Il est alors possible d'accéder à chaque attribut des classes parentes.

```
>>> class Fruit:
...     def __init__(self, color):
...         self.color = color
>>> class Appel(Fruit):
...     pass
>>> a = Appel('red')
>>> a.color
'red'
```

Il est possible de dériver une classe qui est déjà dérivée d'une autre classe, et ce autant de fois que l'on souhaite.

17.3.2 Polymorphisme

17.4 Exercices

1. Écrire une classe qui permettra de gérer un fichier (dont le nom sera passé au constructeur). Elle devra comporter plusieurs méthodes qui réaliseront des opérations sur le fichier :
   - count : renvoie le nombre de lignes, de mots et de lettres.
   - head(n) : renvoie les n premières lignes.
   - tail(n) : renvoie les n dernières lignes.
   - find(str) : afficher toutes les lignes contenant la chaine str.
   - replace(str1, str2) : afficher à l'écran toutes les lignes du fichier, en remplaçant chaque occurrence de str1 par str2.
   - tout ce qui pourra vous passer par la tête (mais qui ne modifie pas le fichier lui-même).

   Il faudra veiller à ne pas charger en mémoire tout le contenu du fichier, ni à le conserver ouvert trop longtemps.
2. Proposer une amélioration du système de droits (exercice 4 du chapitre *Types de base*) en utilisant les classes.

CHAPITRE 18

Erreurs et exceptions

18.1 Introduction

Les erreurs et exceptions ont été mentionnées de nombreuses fois dans ces pages, mais nous nous sommes toujours contenter de dire ce qui les déclenchait (et donc comment les éviter). Même si parfois, les exceptions indiquent un réel problème, elles ne devraient pas forcément interrompre l'exécution du programme.

L'on peut distinguer deux types d'erreurs :
– les erreurs de syntaxe, qui indiquent une mauvaise syntaxe dans l'écriture du programme ;
– les exceptions, qui surviennent lorsqu'un comportement non prévu à l'origine survient.
Dans tous les cas, une pile, appelée *Traceback*, est affichée. Elle permet de retracer les différentes étapes qui ont conduit à l'erreur, ainsi que des informations utiles pour la réparer (le type d'erreur, le ficher, la ligne...).

18.2 Lever une exception

Une exception est levée grâce au mot-clé `raise` : il suffit d'indiquer le nom de l'exception levée, suivie éventuellement d'arguments :

```
raise ExceptionName(args)
```

Voici un exemple simple :

```
>>> raise ValueError('Just a test.')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Just a test.
```

Si vous souhaitez déterminer si une erreur a été levée, mais sans la prendre en charge, il est possible de lever la même erreur dans le bloc `except` à l'aide de `raise`.

```
>>> try:
...     raise Exception('An exception')
... except Exception:
...     print('Here the exception is catched.')
...     raise
Here the exception is catched.
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
Exception: An exception
```

18.3 Gérer les exceptions

18.3.1 Généralités

Les exceptions se gèrent grâce au couple `try-except`. Un bloc `try` doit être accompagné d'au moins un bloc `except` (il n'y a aucune limite maximale). Il existe trois manières d'écrire la ligne `except` :
– en indiquant le nom de l'erreur concernée : `except Error:` ;
– en indiquant un tuple contenant plusieurs erreurs : `except (Error1, ..., Errorn):` ;
– en n'indiquant rien : `except:`.
Ainsi, un ensemble `try-except` s'écrira de la forme :

```
try:
    instructions
except Error1:
    instructions si Error1
except Error2:
    instructions si Error2
except (Error3, Error4):
    instructions si Error3 ou Error4
except:
    instructions si autre erreur
```

Prenons un exemple concret :

```
>>> try:
...     num1 = int(input('Enter a number: '))
...     num2 = int(input('Enter another number: '))
... except ValueError:
...     print('You must enter only numbers.')
```

Imaginons que nous souhaitions demander deux nombres à l'utilisateur, pour ensuite effectuer diverses opérations (par exemple). Comme la fonction `input()` retourne une chaine, nous devons la convertir en entier. Toutefois, si l'utilisateur n'a pas suivi la consigne, et qu'il entre une chaine, alors l'interpréteur ne parviendra pas à la convertir en nombre et lèvera une exception `ValueError`.

Si jamais une exception qui n'était pas attendue était levée, alors le Traceback normal serait affiché (ce qui n'est pas le cas ici).

Notons aussi que le `try` prend en charge toute erreur (du même type que celui spécifié) qui pourrait survenir dans une fonctions appelées dans le bloc `try` (et ce récursivement).

```
>>> def zerodiv():
...     1/0
>>> try:
...     zerodiv()
... except ZeroDivisionError:
...     print('Int division by zero.')
Int division by zero.
```

Note : Il est tout à fait possible de ne préciser aucune erreur après `except`, comme on l'a vu, mais il s'agit d'une pratique fortement déconseillé :
– l'on perd en souplesse, puisque l'on ne peut gérer chaque erreur au cas par cas ;

– cela masque toutes les erreurs qui surviennent, même celles que l'on imaginait pas, ce qui a pour conséquence de complexifier largement le débuggage.

18.3.2 Blocs évolués

Au même titre que pour les boucles, il est possible d'adjoindre un bloc `else` aux blocs de gestion d'exceptions : ce dernier ne sera exécuté que si aucune exception n'a été levée.

Il est aussi possible d'ajouter un bloc `finally` : les instructions de ce dernier bloc seront exécutées quoiqu'il arrive. Ce dernier peut être utilisé pour les actions de "nettoyage" (fermeture d'un fichier ouvert dans `try`, fermeture d'une base de données...).

Note : `with` peut parfois remplacer avantageusement `finally`.

Voici un exemple général (testez-le en entrant une chaine puis un nombre) :

```
>>> try:
...     num = int(input('Enter a number: '))
... except ValueError:
...     print('You must enter a number.')
... else:
...     print('You have enter a number.')
... finally:
...     print('This text is always printed.')
```

Note : Il est important de noter que le bloc `finally` sera *toujours* exécuté, même si un `return`, `break` ou `continue` intervient *avant* (dans le code).

```
>>> def func():
...     try:
...         return 1
...     except:
...         pass
...     finally:
...         print(2)
>>> func()
2
1
```

18.3.3 Variable de l'erreur

Il est aussi possible de stocker l'erreur dans une variable, afin de la manipuler et d'obtenir des informations plus précises : la variable spécifiée dans la ligne `except` est alors associée à l'exception qui a été levée. Il est alors possible d'accéder aux attributs et aux arguments de l'exception concernée.

```
>>> try:
...     raise Exception('First argument', 'Another argument')
... except Exception as error:
...     print(type(error))
...     print(error.args)
<class 'Exception'>
('First argument', 'Another argument')
```

18.4 Définition d'exceptions

Les exceptions, comme tout élément en Python, est un objet. Il est donc très simple de définir des exceptions personnalisées : il suffit de dériver une classe d'exception pré-existante.

```
>>> class MyError(Exception):
...     pass
>>> raise MyError
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.MyError
```

Note : Il aurait bien entendu été possible de redéfinir la méthode `__init__()`, ou encore d'ajouter d'autres méthodes et attributs. Toutefois, les classes d'exception sont souvent simplifiées autant que possible.

18.5 Exceptions built-in

Au même titre que pour les fonctions, Python contient tout un set d'exceptions built-in. Voici donc une liste des exceptions les plus courrantes :

CHAPITRE 19

Entrées et sorties

19.1 Affichage de données (`print()`)

La fonction `print()` permet d'afficher du texte. L'utilisation la plus simple que l'on puisse en faire est de passer en argument [1] la variable à afficher : `print(variable)`.

```
>>> print('a variable')
a variable
>>> print(10)
10
>>> var = 'hello'
>>> print(var)
hello
```

Il est possible de donner plusieurs arguments à la suite :

```
>>> print(1, 'str', 1+2j)
1 str (1+2j)
```

On peut modifier le symbole qui les sépare en renseignant l'argument `sep` (par défaut il s'agit d'un espace) :

```
>>> print(1, 'str', 1+2j, sep='_')
1_str_(1+2j)
```

Enfin, il est aussi possible de choisir le symbole en fin de ligne (par défaut c'est un fin de ligne) :

```
>>> print('first print', end=' '); print('second print')
first print second print
```

Pour afficher une nouvelle ligne, il suffit d'appeler la fonction sans argument :

```
>>> print()

>>>
```

Enfin, on peut spécifier le fichier où est écrit l'argument à l'aide de l'argument `file`.

1. L'utilisation et le vocabulaire des fonctions seront détaillés dans le chapitre *Fonctions*.

19.2 Entrée au clavier (`input()`)

La fonction `input()` permet de lire une ligne écrite et de la stocker dans une variable. La saisie de la ligne est interrompue avec la touche `entrée`.

Elle accepte un argument optionnel qui permet d'afficher une chaine de caractères (par exemple, pour donner des informations) :

```
>>> input()
Two words
'Two words'
>>> input()
1
'1'
>>> prompt = input()
Text
>>> prompt
'Text'
>>> name = input('Enter your name: ')
Enter your name: Harold
>>> name
'Harold'
```

Warning : Le type de la variable créée sera toujours une chaine de caractères. Il faudra donc prendre garde à effectuer les tests et conversions nécessaires avant son utilisation.

19.3 Fichiers

19.3.1 Ouverture et fermeture

On ouvre un fichier à l'aide de la fonction `open()` :

open(*'file'*, *'mode'*)

Les deux arguments doivent être des chaines de caractères. Le mode peut être :
– `r` : lecture seule ;
– `w` : écriture ;
– `a` : ajout.
Le fichier est créé s'il n'existe pas avec les options `w` et `a`. Avec ces modes, il n'est pas possible de lire ce qui est dans le fichier.

La méthode `close()` permet de fermer un fichier.

19.3.2 Méthodes de fichier

read()
: Lit tout le contenu du fichier et le renvoie sous forme d'une chaine de caractères.

readline()
: Lit la ligne suivant le curseur dans le fichier et la renvoie sous forme d'une chaine de caractères.

readlines()
: Lit tout le contenu du fichier et retourne les lignes sous forme d'une liste de chaines de caractères,.

write(*'string'*)
Écrit la chaine `string` dans le fichier.

writelines(*sequence*)
Écrit la séquence `sequence` dans le fichier, en mettant bout à bout les éléments.

Rien ne sera écrit dans le fichier avant l'appel de la méthode `close`'().

19.3.3 Exemple

```
>>> names = ['Yves', 'Jean', 'Dupont']
>>> file = open('names.txt', 'w')
>>> for name in names:
...       output = file.write(name + '\n')
>>> file.close()
```

En ouvrant le fichier `names.txt`, vous pourrez voir un nom sur chaque ligne. On aurait pu utiliser la méthode `writelines()` plutôt que de passer par une boucle.

```
>>> file = open('names.txt', 'r')
>>> for line in file.readlines():
...     print(line)
Yves

Jean

Dupont
```

Warning : Il peut être nécessaire de supprimer les signes de nouvelles `\n` avant d'afficher les chaines de caractères. De même, il peut être nécessaire de les ajouter si l'on désire écrire plusieurs chaines sur des lignes différentes.

Une utilisation plus générale des fichiers et des dossiers sera abordée dans le chapitre *Gestion des fichiers et dossiers*.

19.4 Exercices

1. Écrivez un programme qui demande son nom à l'utilisateur, et qui l'affiche ensuite à l'écran avec un message d'accueil.
2. Écrire un programme qui demande deux nombres à l'utilisateur, et qui affiche ensuite la somme, la différence et la multiplication de ces deux nombres.
3. Améliorez le programme précédent en ajoutant la division. Faites attention à prendre en compte le cas où le deuxième nombre est nul.
4. Écrire un programme qui récupère une liste de noms dans un fichier (un nom par ligne) et qui les trie par ordre alphabétique.

CHAPITRE 20

Modules

20.1 Introduction

L'utilisation interactive de Python n'est pas très pratique, car elle ne permet pas de sauvegarder les variables et autres résultats. Pour y remédier, l'on écrit le code dans des fichiers [1], appelés modules (ou scripts), qui permettent de réutiliser le code. De même, il est fortement conseillé de diviser un long programme en de nombreux modules pour plus de clarté.

Un module peut contenir aussi bien des définitions (de classe ou de fonctions) que des expressions, mais ces dernières ne seront exécutées qu'une seule fois (la première fois que le module est importé).

Les modules peuvent être nommés de la même manière que les variables. Un module est importé grâce au mot-clé `import` suivi du nom d'un ou plusieurs modules [2] :

```
import mod1, ..., modn
```

Il est alors possible d'accéder aux éléments qu'il contient grâce au point (puisqu'un module est considéré comme un objet). Tout module peut être importé dans un autre module, dans le module principal ou dans l'interpréteur interactif.

Enfin, au même titre que pour les fonctions et le classes, un module peut comporter une docstring.

Créons par exemple un fichier `fibo.py` contenant du code :

```
''' This module contains the Fibonacci series.
'''

__all__ = ['fibo_func']

def fibo_func(n):
    a, b = 0, 1
    while b < n:
        yield b
        a, b = b, a + b

func_type = "generator"

if __name__ == '__main__':
    print(list(fibo_func(6)))
```

1. Par convention, ces fichiers portent l'extension `.py`.
2. Il est commun de placer tous les imports au début du fichier (mais après la docstring), même si ce n'est pas obligé.

Maintenant, lançons l'interpréteur interactif :

```
>>> import fibo
>>> print(fibo.func_type)
generator
>>> list(fibo.fibo_func(4))
[1, 1, 2, 3]
```

Il est possible de définir des *alias*[3] :

```
>>> fib = fibo.fibo_func
>>> list(fib(4))
[1, 1, 2, 3]
```

Dans chaque module importé est définie la variable `__name__`, qui contient le nom du module :

```
>>> fibo.__name__
'fibo'
```

Terminons cette section en notant qu'un module n'est importé qu'une seule fois. Les modifications apportées après l'import ne seront donc pas prises en compte, à moins que l'on redémarre l'interpréteur ou que l'on relance le programme.

20.2 Variantes

L'on peut choisir d'importer certaines éléments d'un module directement dans le contexte global, en utilisant `from-import`. L'on peut attribuer un *alias* à un élément apporté à l'aide du mot-clé `as`. Il est possible de mixer les éléments avec alias et sans alias.

    from module import name1 [as alias1], ..., namen [as aliasn]

Dans ce cas là, il convient de noter que le module lui-même n'est pas importé.

```
>>> from fibo import fibo_func, func_type as type
>>> list(fibo_func(4))
[1, 1, 2, 3]
>>> type
'generator'
>>> fibo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'fibo' is not defined

>>> from fibo import fibo_func as fib
>>> list(fib(4))
[1, 1, 2, 3]
```

L'étoile `*` permet d'importer tout un ensemble d'éléments, définis dans une liste nommée `__all__`.

Reprenons le fichier précédent et ajoutons au début de ce dernier :

```
__all__ = ['fibo_func']

>>> from fibo import *
>>> list(fibo_func(4))
[1, 1, 2, 3]
```

3. Ceci n'est pas la manière habituelle de procéder. Nous verrons par la suite comment créer un véritable *alias*.

Toutefois, cette syntaxe n'est pas autorisée en dehors du niveau du module (par exemple elle ne peut être utilisée dans la définition d'une fonction).

20.3 Scripts

Pour utiliser un module comme un script, il suffit de fournir, dans un shell, le nom du module en argument :

```
$ python fibo.py
```

Dans ce cas uniquement, la variable `__name__` vaut `__main__`. Ceci sert généralement à introduire, en fin de module, du code qui ne doit être exécuté que si le module est utilisé comme un script. Par exemple, nous pourrions ajouter à la fin de notre fichier `fibo.py` :

```
if __name__ == '__main__':
    print(list(fibo_func(6)))
```

Le résultat de sera évidemment `[1, 1, 2, 3, 5]`.

20.4 Mécanismes

20.4.1 Module path

Lorsque l'on importe un module, ce dernier est recherché dans les dossiers contenus dans la variable `sys.path` (une liste, que l'on peut modifier). Cette dernière contient les mêmes éléments que la variable shell `PYTHONPATH` (dont la syntaxe est la même que pour `PATH`) ainsi que les éventuels répertoires ajoutés.

Ainsi, si l'un de vos modules ne veut pas être importé, vérifiez s'il est bien dans le path.

Warning : Un module ne devrait pas être nommé de la même manière qu'un module standard.

20.4.2 Fichiers compilés

L'interpréteur Python crée, lorsqu'il importe un module (nommé `name.py` pour l'exemple), un fichier nommé `name.pyc`. Il s'agit d'une version compilée, et donc plus rapide, du fichier en question.

Ainsi, lorsqu'il existe à la fois une version `.py` et une version `.pyc`, l'interpréteur vérifie si le code contenu dans le `.py` n'a pas changé depuis la dernière fois qu'il a été importé ; le cas échéant, l'interpréteur utilisera la version compilée. Par contre, si le code a changé, alors l'interpréteur chargera la version `.py` et créera un nouvel `.pyc`, pour un usage ultérieur.

Note : Il est aussi possible de générer du bytecode optimité (l'extension des fichiers sera alors `.pyo`). Ceci sera vu en détails plus tard.

20.4.3 Mise en garde sur l'import

Lorsqu'un module est importé, une référence est stockée dans le dictionnaire `sys.modules`.

Par exemple, au lancement de l'interpréteur interactif, j'obtiens :

```
>>> from sys import modules
>>> modules
{'__main__': <module '__main__' (built-in)>, 'site': <module 'site' from '/usr/local/lib/python3.1/s
```

Cela signifie qu'un module n'est importé qu'une fois : les fois suivantes, Python ne fait rien. Ainsi, il évite d'importer plusieurs fois des modules ce qui permet d'améliorer les performances. Toutefois, cela est peu pratique lorsque l'on veut tester un module, car il est nécessaire de quitter l'interpréteur et de le relancer si on édite le fichier. Pour remédier à ce problème, on pourra se tourner vers l'interpréteur *IPython*.

20.5 Modules standards

Il existe de nombreux modules distribués par défaut avec Python, dont il serait trop long de faire la liste. Pour cette raison, la liste qui suit contient uniquement les principaux modules.
– `sys` : de nombreuses informations sur le système ;
– `os` : fonctions pour interragir avec le système d'exploitation ;
– `os.path` : fonctions pour manipuler fichiers et dossiers ;
– `glob` : ce module contient une fonction pour créer une liste de fichiers au moyen de jokers (`*`) ;
– `re` : utilisation des expressions régulières ;
– `math` : fonctions et constantes mathématiques (sin, cos, exp, e, pi, etc) ;
– `random` : génération de nombres aléatoires ;
– `datetime` : utilisation de dates.
Il ne s'agit que d'un très bref aperçu, et il est conseillé au lecteur de se renseigner dans la documentation dès qu'il a besoin d'une certaine fonction.

20.6 Packages

Un package est un ensemble de plusieurs modules. Il s'agit donc d'un dossier qui, pour que Python sache qu'il s'agit d'un package, doit contenir un fichier `__init__.py`[4].

L'on accède au module d'un package en utilisant le point, et il est possible de créer autant de sous-packages que l'on souhaite. Nous pourrions donc avoir une arborescence de ce type :

```
sound/
    __init__.py
    formats/
        __init__.py
        wavread.py
        wavwrite.py
        ...
    effects/
        __init__.py
        echo.py
        surround.py
        ...
    filters/
        __init__.py
        equalizer.py
        vocoder.py
```

Nous pouvons alors importer tout un package, un ou plusieurs modules individuellement.

4. Ce fichier est généralement vide, mais il peut contenir du code qui sera exécuté lors de l'import du package, comme par exemple définir la variable `__all__`.

```
>>> import sound.filters.echo
>>> # access at the elements with "sound.filters.echo"
>>> from sound.filters import echo
>>> # access at the elements with "echo"
>>> from sound.effects.echo import echofilter
>>> # access at the function "echofilter"
```

20.7 Imports relatif et absolus

Il est possible d'accéder à un module aussi bien grâce au chemin absolu, ou bien grâce au chemin relatif si ce dernier se trouve dans le même dossier (ou donc dans un sous-dossier).

Par exemple, si l'on reprend la liste de fichiers précédentes, dans le fichier `echo.py`, nous pourrions importer le module `surround` grâce à l'une des deux syntaxes suivantes :

```
import sound.effects.surround
import surround
```

Quatrième partie

Aspects spécifiques

CHAPITRE 21

Expressions régulières

21.1 Introduction

Les expressions régulières (ou rationnelles), communément abrégé en regex, constituent un outil fondamental lorsqu'il s'agit de rechercher/remplacer des caractères d'une manière complexe.

Les regex elles-mêmes demanderaient un livre entier pour être maitrisé, aussi conseillerai-je de se reporter à des livres ou des sites spécialisés sur le sujet, même si je donnerai les bases pour s'en servir et comprendre leur intérêt.

Une expression régulière consiste en une chaine de caractères, souvent appelée "motif" [1]. Elle sera donc écrite entre guillemets. Un `r` placé devant la chaine permet de considérer l'antislash \ comme un caractère normal. Par exemple, on pourra écrire :

```
>>> regex = r'a0\'
```

Note : OpenOffice permet de faire des recherches par regex (*Rechercher → Autres options*). Cela peut être utile pour s'entrainer à manipuler les regex.

21.2 Syntaxe

Les regex répondent à une syntaxe très codifiée et possèdent de nombreux symboles ayant un sens particulier.

Pour débuter, tout caractère alphanumérique n'a pas de signification spéciale : `A` correspond simplement à la lettre A, `1` au chiffre 1, etc.

Quant aux principaux symboles spéciaux, il sont :
- `.` : désigne n'importe quel caractère ;
- `^` : indique que le début de la chaine doit correspondre (c'est à dire qu'une chaine ne pourra correspondre que si elle commence de la même manière, qu'elle est précédée d'espaces ou d'un saut de ligne) ;
- `$` : indique que la fin de la chaine doit correspondre (la même remarque que précédemment s'applique, mais au niveau de la fin) ;
- `{n}` : indique que le caractère précédent doit être répété `n` fois.
- `{n,m}` : indique que le caractère précédent doit être répété entre `n` et `m` fois.

1. Ou encore "pattern" en anglais.

– `*` : le caractère précédent peut être répété aucune ou plusieurs fois. Par exemple, à `ab*` peuvent correspondre : `a`, `ab`, ou `a` suivi d'un nombre quelconque de `b`.
– `+` : le caractère précédent peut être répété une ou plusieurs fois. Par exemple, à `ab+` correspond un `a` suivi d'un nombre quelconque de `b`.
– `?` : le caractère précédent peut être répété zéro ou une fois. Par exemple, à `ab?` correspondent `ab` et `a`.
Les quatres derniers symboles sont dits "gourmands"[2] : cela signifie qu'ils chercheront un maximum de caractères, ce qui peut parfois poser problème. Pour éviter ce comportement, on peut rajouter un `?` après ces derniers, c'est à dire utiliser : `??`, `+?` et `*?`.

L'antislash permet d'échapper tous ces caractères spéciaux.

Les crochets `[]` permettent d'indiquer une plage de caractère, par exemple `[e-h]` correspondra à `e`, `f`, `g` ou `h`. Les parenthèses permettent de grouper certaines expressions[3].

Finalement, il reste quelques caractères spéciaux assez utiles :
– `\w` : il correspond à tout caractère alphabétique, c'est à dire qu'il est équivalent à `[a-zA-Z]` ;
– `\W` : il correspond à tout ce qui n'est pas un caractère alphabétique ;
– `\d` : il correspond à tout caractère numérique, c'est à dire qu'il est équivalent à `[0-9]` ;
– `\D` : il correspond à tout ce qui n'est pas un caractère numérique.

21.3 Utilisation

Le module s'appelle `re`.

Il existe deux manières d'utiliser les regex :

1. La première consiste à appeler la fonction avec en premier paramètre le motif, et en deuxième paramètre la chaine à analyser.
2. La seconde consiste à compiler la regex, et à ensuite utiliser les méthodes de l'objet créer pour analyser une chaine passée en argument. Cette méthode permet d'accélérer le traitement lorsqu'une regex est utilisée plusieurs fois.

La plupart des fonctions et méthodes prennent des arguments optionnels, appelés drapeau ("flag" en anglais) qui permettent de modifier le traitement de l'objet. Le drapeau le plus intéressant est `re.I` (ou `re.IGNORECASE`) qui permet d'ignorer la casse lors de la recherche.

`re.`**`compile`**(*pattern*)
Compiler une expression régulière et retourne un objet contenant les mêmes méthodes que les fonctions décrites juste après.

`re.`**`search`**(*pattern*, *string*)
Cherche le motif dans la chaine passée en argument et retourne un `MatchObject` si des correspondances sont trouvées, sinon retourne `None`.

`re.`**`split`**(*pattern*, *string*)
Découpe la chaine `string` selon les occurences du motif.

```
>>> re.split(r'\W', 'Truth is beautiful, without doubt.')
['Truth', 'is', 'beautiful', '', 'without', 'doubt', '']
```

`re.`**`findall`**(*pattern*, *string*)
Retourne toutes les sous-chaines de `string` correspondant au motif.

`re.`**`sub`**(*pattern*, *repl*, *string*)
Retourne la chaine `string` où le motif a été remplacé par `repl`.

2. "Greedy" en anglais.
3. Ce qui peut permettre de s'y référer par la suite.

Le principal intérêt est d'utiliser la fonction `match()` afin de vérifier si une chaine se trouve dans une autre ou si elle respecte un certain format (ce qui peut être utilisé pour vérifier si un numéro de téléphone ou une adresse email à un format correct).

Pour tout autre utilisation des MatchObject, je renvoie le lecteur à la documentation.

La séquence

```
prog = re.compile(pattern)
result = prog.match(string)
```

est équivalente à

```
result = re.match(pattern, string)
```

21.4 Exemples

```
>>> import re
>>> # trouver tous les adverbes en -ment
>>> text = "Il s'était prudemment déguisé mais fut rapidement capturé par la police."
>>> re.findall(r"\w+ment", text)
['prudemment', 'rapidement']
```

21.5 Liens

– Regular Expression HOWTO par A.M. Kuchling : il s'agit d'un document utilisant une ancienne version de Python mais toujours adaptée pour apprendre les regex.
– Python Regular Expression Testing Tool : ce site permet de tester des regex en utilisant un site particulier.
– Dive Into Python 3 : Chapter 5.

21.6 Exercices

1. Écrire un motif pour les numéros de téléphone, en prenant garde aux différentes manières de les écrire (avec un point, un tiret ou un espace entre les chiffres). On se limitera à des numéros français dont les numéros sont groupés par deux ou pas du tout.
2. Utiliser le motif précédent qui extrait tous les numéros de téléphone d'un texte et qui les affiche à l'écran, sous une forme normalisée (par exemple tous les chiffres sont groupés par deux, et le séparacteur est un point).

CHAPITRE 22

Gestion des fichiers et dossiers

Les principaux modules utilisés sont `os` et `os.path`. Pour cette raison, je supposerai que ces modules auront été importés dans tous les exemples qui suivent.

22.1 Introduction aux notions de fichiers et de dossiers

Presque n'importe qui saurait faire la différence entre un fichier — qui contient des informations (texte, musique, vidéo, image... pour les plus connus) — et un dossier (ou répertoire) — qui contient d'autres dossiers ou des fichiers. Toutefois, si l'on veut comprendre comment les langages de programmation (et donc aussi les systèmes d'exploitation) les traitent, il faut aller un peu plus loin. Je n'introduirai que les notions de base, puisque ce n'est pas le sujet principal de ce livre.

Le système de fichiers est donc organisé hiérarchiquement : on parle d'arborescence.

22.1.1 Chemins

Sur un ordinateur un chemin [1], représente une suite de noms de dossier (avec éventuellement un fichier à la fin) qui indique commencer arriver à un endroit précis. Voici quelques exemples de chemins :

```
/home/harold
C:\Python31\python.exe
```

Nous remarquons une différence flagrante entre ces deux chemins : dans le premier, le *séparateur* est un slash `/` — typique des systèmes Unix [2] —, tandis qu'il s'agit d'un antislash `\` dans le second — propre au système Windows [3]. Heureusement pour nous, nous n'aurons pas à faire la différence car Python comprend même lorsque nous utilisons des slashs sous Windows ; ainsi nous pourrons écrire sans problèmes :

```
C:/Python31/python.exe
```

Il existe deux types de chemins :
– les chemins *absolus* : ces derniers partent de la racine du système (`/` sous Unix, `C:` sous Windows), et indiquent le chemin détaillé pour arriver à l'endroit indiqué (les deux exemples donnés plus haut sont des chemins absolus) ;

1. "Path" en anglais. Ce terme est très souvent utilisé.
2. Les urls sont aussi des chemins vers des fichiers et dossiers d'un système Unix, d'où l'écriture identique.
3. Généralement, lorsque le chemin se termine par un dossier, on met quand même le séparateur à la fin.

– les chemins *relatifs* : le chemin est donné à partir d'un dossier spécifique, qui est considéré, en quelque sorte, comme une nouvelle racine. Par exemple

```
python.exe
```

est un chemin relatif et n'a de sens que si l'on précise le dossier courant (celui où l'on est) ; ici il s'agira donc du dossier `C:\Python31`.

Il est très important de faire la différence entre ces deux types de chemins. Généralement, nous préférerons utiliser des chemins absolus dans nos programmes, afin d'être certain de l'endroit où l'on travaille (par contre, lors de nos tests, nous utiliserons des chemins relatifs).

Il est facile de distinguer les chemins relatifs et absolus : un chemin absolu commencera *toujours* par un slash sous Unix, ou par une lettre de lecteur suivie de deux points sous Windows.

Sous Unix, deux dossiers particuliers sont définis dans chaque répertoire :
– `.` désigne le répertoire courant.
– `..` désigne le répertoire supérieur.
Pour fixer les idées, imaginons que je me trouve dans le dossier `/home/harold/python/livre`. Alors `..` sera équivalent au dossier `/home/harold/python` et `.` désignera `/home/harold/python/livre`.

22.2 Manipulation des noms chemins

En Python, les chemins sont tout simplement des objets de type str. Il est donc possible de les manipuler avec les opérations classiques des chaines, bien que cela ne soit pas toujours conseiller (par exemple, pour coller deux chemins, il vaut mieux utiliser `join()` que l'opérateur +). Toutes les fonctions de base pour manipuler les noms de chemin se trouvent dans le module `os.path`.

`os.path.`**`abspath`**(*path*)
Retourne le chemin absolu du chemin relatif `path` donné en argument.
Il permet aussi de simplifier les éventuelles répétitions de séparateurs.

`os.path.`**`realpath`**(*path*)
Retourne le chemin `path` en remplaçant les liens symboliques par les vrais chemins, et en éliminant les séparateurs excédentaires.

`os.path.`**`split`**(*path*)
Sépare `path` en deux parties à l'endroit du dernier séparateur : la deuxième partie contient le fichier ou le dossier final, et la première contient tout le reste (c'est à dire le chemin pour y accéder).

Type retourné 2-tuple of strings

`os.path.`**`basename`**(*path*)
Retourne le dernier élément du chemin. Cela revient à appeler :func :`~os.path.split` et à récupérer le deuxième élément du tuple.

`os.path.`**`dirname`**(*path*)
Retourne le chemin pour accéder au dernier élément. Cela revient à appeler :func :`~os.path.split` et à récupérer le premier élément du tuple.

`os.path.`**`expanduser`**(*path*)
Si path est ~ ou `~user`, retourne le chemin permettant d'accéder au dossier personnel de l'utilisateur.

`os.path.`**`join`**(*path1*[, *path2*[, ...]])
Permet de concaténer plusieurs chemins.

Warning : Toutes ces fonctions retournent une valeur même si le chemin donné en argument n'existe pas. Il peut donc être nécessaire de tester leur existence avec les fonctions que nous verrons plus bas.

Voici quelques exemples de manipulations de noms de chemin :

```
>>> path = '/home/harold/Musique'
>>> os.path.split(path)
('/home/harold', 'Musique')
>>> os.path.join(path, 'Classique')
'/home/harold/Musique/Classique'
>>> os.path.expanduser('~')
'/home/harold'
>>> os.path.abspath('.')
'/mnt/data/Projets/Enseignements/Python'
>>> os.path.realpath('/mnt//dev/')
'/mnt/dev'
>>> os.path.dirname(path)
'/home/harold'
>>> os.path.basename(path)
'Musique'
```

Warning : Observez le code suivant :

```
>>> path = '/home/harold/Musique/'
>>> os.path.split(path)
('/home/harold/Musique', '')
```

Il faut donc faire attention à ne pas laisser de séparateur à la fin. On pourra utiliser `realpath()` ou `abspath()` avant tout appel à une autre fonction de chemins, afin de s'assurer que les chemins sont au bon format :

```
>>> path = '/home/harold/Musique/'
>>> os.path.split(os.path.abspath(path))
('/home/harold', 'Musique')
```

22.3 Lister les fichiers et les dossiers

Le module `os` contient une fonction qui permet de lister simplement les éléments d'un dossier.

`os.`**`listdir`**(*path*)
: Liste tous les dossiers et fichiers qui se trouvent dans `path`. Les dossiers `.` et `..` sont automatiquement retirés.

Toutefois, pour un usage plus compliqué, on préfera le module `glob`, qui contient deux fonctions uniquement. Les fonctions de ce module permettent de lister tous les fichiers dont les noms correspondent au motif donné en argument. Le motif peut contenir certains caractères spéciaux :
– `*` remplace n'importe quelle séquence de caractères.
– `?` remplace un caractère.
– `[]` symbolise n'importe quel caractère indiqué dans les crochets.

`glob.`**`glob`**(*pathname*)
: Liste tous les dossiers et fichiers dont le motif du nom correspond à `pathname`.

`glob.`**`iglob`**(*pathname*)
: Fonctionne de même que `glob()` mais retourne un itérateur. Cette fonction est donc préférable.

L'exemple qui suit est tiré de la documentation Python. Il suppose que vous ayez un dossier qui contient au moins les fichiers suivants : `1.gif`, `2.txt` et `card.gif`.

```
>>> import glob
>>> glob.glob('./[0-9].*')
['./1.gif', './2.txt']
>>> glob.glob('*.gif')
['1.gif', 'card.gif']
>>> glob.glob('?.gif')
['1.gif']
```

22.4 Tests sur les fichiers et les dossiers

Les tests de base sur les dossiers et fichiers se font tous à partir de fonctions du module `os.path`.

`os.path.`**`exists`**(*path*)
: Permet de vérifier si `path` est un chemin réel.

`os.path.`**`isabs`**(*path*)
: Teste si `path` est un chemin absolu.

`os.path.`**`isdir`**(*path*)
: Teste si `path` pointe vers un dossier.

`os.path.`**`isfile`**(*path*)
: Teste si `path` pointe vers un fichier.

Voici quelques exemples :

```
>>> os.path.exists(path)
True
>>> os.path.isabs(path)
True
>>> os.path.isfile(path)
False
```

22.5 Autres opérations

Dans cette section nous étudierons les fonctions du module `os` qui permettent de créer, supprimer et renommer les fichiers et répertoires.

`os.`**`mkdir`**(*path*)
: Crée le dossier à la fin de `path`. Si l'un des dossiers intermédiaires n'existe pas, une exception `OSError`'.

`os.`**`makedirs`**(*path*)
: Cette fonction est équivalente à un appel récursif à `mkdir()` : elle crée récursivement tous les dossiers qui n'existe pas jusqu'au dossier final de `path`.

`os.`**`remove`**(*path*)
: Supprime le fichier ou le dossier pointé par `path`.

`os.`**`rename`**(*old*, *new*)
: Renomme `old` en `new`.

22.6 Exercices

1. Créer un programme qui liste uniquement les fichiers d'un répertoire et qui les ordonne par taille croissante.

CHAPITRE 23

Système

Ce chapitre est fortement lié au précédent, et c'est pour cette raison qu'il sera très court : nous nous contenterons d'aborder les dernières fonctions utiles du module `os` ainsi que celle du module `sys`.

23.1 Informations sur Python

Le module `sys` permet d'obtenir des informations sur certaines propriétés de la version Python qui est utilisée, ainsi que sur son comportement.

`sys.`**`path`**
: Liste des dossiers dans lesquels Python cherche les modules à importer (voir *Modules*).

`sys.`**`modules`**
: Liste des modules importés par le programme (voir *Modules*).

`sys.`**`version`**
: Donne des informations sur la version de Python qui est utilisée.

`sys.`**`version_info`**
: Donne, sous forme d'un tuple, des informations sur la version de Python. Cette forme peut être plus pratique à utilisée lorsqu'il s'agit de tester si la version utilisée correspond à certains critères.

```
>>> if sys.version_info[0] < 3:
...     print('La version de Python que vous utilisez est trop ancienne.')
```

`sys.`**`float_info`**
: Donne, sous forme d'un objet, des informations sur les nombres à virgule du système (précision, taille maximale…).

`sys.`**`getfilesystemencoding`**`()`
: Renvoie le nom de l'encodage utilisé par le système.

Voici quelques exemples :

```
>>> sys.version
'3.1.1 (r311:74480, Feb  5 2010, 20:43:13) \n[GCC 4.4.2 20091222 (Red Hat 4.4.2-20)]'
>>> sys.version_info
sys.version_info(major=3, minor=1, micro=1, releaselevel='final', serial=0)
```

23.2 Informations sur le système

Le module `os` permet d'accéder à de nombreuses informations sur le système.

`os.name`
: Indique le nom du système. Les principales possibilités sont : "posix" (Linux, Mac), "nt" (Windows).

`os.getcwd()`
: Indique le répertoire de travail, c'est à dire, en quelque sorte, l'endroit où sont exécutés les scripts : tout chemin relatif partira de ce dernier. Il est possible de le changer avec `chdir()`.

`os.uname()`
: Renvoie un 5-tuple contenant : le nom du système, le nom de la machine, la version du système et le type d'architecture.

Voici quelques exemples :

```
>>> os.name
'posix'
>>> os.getcwd()
'/mnt/data/Projets/Enseignements/Python'
>>> os.uname()
('Linux', 'harold-laptop', '2.6.32.9-70.fc12.i686.PAE', '#1 SMP Wed Mar 3 04:57:21 UTC 2010', 'i686')
```

23.3 Arguments en ligne de commande

Lorsque l'on crée un programme, il peut être très utile de lui donner des informations supplémentaires dès le lancement afin de modifier son comportement ou pour éviter d'utiliser `input()`. Ceci peut être réalisé en utilisant des arguments de ligne de commandes : il s'agit simplement de texte (qui peut représenter un nom de fichier, un nombre, une chaine...) qui se place après le nom du fichier, c'est à dire, dans un terminal, on écrira :

```
$ python file.py arg1 arg2 arg3 ...
```

Il est ensuite possible de récupérer les arguments grâce à l'attribut `sys.argv`[1] : il s'agit d'une liste qui contient tous les arguments (le premier élément étant le nom du fichier).

Warning : Les éléments de `sys.argv` sont toujours des chaines de caractères.

On peut imaginer, par exemple, vérifier si l'argument `-v` est présent pour activer le mode verbeux et aussi donner la possibilité de donner un nom de fichier pour sauvegarder dans ce dernier le résultat du script.

```
'''
Le code suivant affiche la somme des cinq premiers entiers. Si le mode
verbeux est activé, alors il affiche les résultats intermédiaires. Si un
nom de fichier est donné, il écrira le résultat dans ce fichier, sinon
il l'affichera à l'écran.
'''

from sys import argv

if '-v' in argv:
    VERBOSE = True
else:
    VERBOSE = False
```

1. `sys.argc` permet d'obtenir le nombre d'arguments. Cela revient au même que `len(sys.argv)`.

```
if '-f' in argv:
    # on récupère l'index de l'élément qui suit "-f".
    index = argv.index('-f') + 1
    FILE = argv[index]
else:
    FILE = None

s = 0

for i in range(6):
    s += i
    if VERBOSE is True:
        print(s)

if FILE is None:
    print('Le résultat est :', s)
else:
    result_file = open(FILE, 'w')
    result_file.write(str(s))
    result_file.close()
    print('Le résultat a été sauvegardé dans :', FILE)
```

Testons notre code :

```
$ python3.1 commandline.py
Le résultat est : 15
$ python3.1 commandline.py -v
0
1
3
6
10
15
Le résultat est : 15
$ python3.1 commandline.py -f fichier.txt
Le résultat a été sauvegardé dans : fichier.txt
```

Le module `optparse` permet une gestion beaucoup plus poussée des arguments passé en ligne de commande.

23.4 Exercices

1. Corriger le programme donner en exemple dans la dernier section afin d'ajouter une gestion des exceptions : par exemple prendre en compte le cas où `-f` n'est pas suivi d'un nom de fichier. On pourra aussi vérifier si le dossier où l'on veut sauvegarder le fichier existe.
2. Corriger le script précédent afin de permettre à l'utilisateur d'entre le nombre `n` d'entiers qu'il veut sommer.

CHAPITRE 24

Classes étendues

Dans ce chapitre, nous aborderons les différentes techniques qui permettent de définir des classes ayant un comportement spécifique.

24.1 Itérateurs

24.1.1 Introduction

Un itérateur est un objet qui peut être parcouru par une boucle `for`. De nombreux objets Python sont en réalités des itérateurs, et leur utilisation permet de simplifier d'une manière élégante le code. De plus, les itérateurs permettent de gagner en performance : en effet, les éléments de ces derniers ne sont évalués que lorsqu'ils sont nécessaires et utiles. Ainsi, contraitement à une séquence qui est gardée toute entière en mémoire (ce qui peut réduire à grande consommation de mémoire), les itérateurs prennent peu de place. Enfin, les itérateurs peuvent être "infinis".

De nombreuses fonctions, comme `map()`, `sum()` ou encore `max()`, acceptent des itérateurs en argument.

Les itérateurs les plus basiques sont les tuples et les listes :

```
>>> for i in (1, 2):
...     print(i)
1
2
>>> for i in ['a', 'b']:
...     print(i)
a
b
```

Les dictionnaires eux aussi sont des itérateurs, ce qui permet d'écrire

```
>>> dic = {'a': 1, 'b': 2}
>>> for key in dic:
...     print(key)
```

bien qu'une idée plus intuive aurait pu être d'itérer sur la liste des clés en utilisant la méthode `keys()` (qui retourne désormais aussi un itérateur depuis la version 3) :

```
>>> for key in dic.keys():
...     print(key)
```

Les fichiers eux-mêmes sont des itérateurs : les syntaxes `for line in f` et `for line in f.readlines()` ont le même effet, bien que la première soit plus efficace.

La fonction `iter()` permet de transformer un objet en itérateur, tandis que `next()` permet d'obtenir l'élément suivant.

24.1.2 Implémentation

Une classe peut devenir un itérateur à condition d'implémenter les deux méthodes `__iter__` et `__next__`.

Comme exemple, construisons une classe qui prend en argument une séquence et, qui, lorsque utilisée en tant qu'itérateur, retourne les éléments de cette séquence en ordre inverse :

```
class Reverse:
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

L'exception `StopIteration` permet d'indiquer que l'itération est terminée et que la boucle doit s'arrêter. Cette exception est automatiquement interceptée par la boucle `for`.

```
>>> for char in Reverse('Jour'):
...     print(char)
r
u
o
J
```

24.1.3 Module `itertools`

Ce module contient de nombreuses fonctions utiles pour travailler avec des itérateurs. Citons par exemple la fonction `chain()` qui permet de "coller" deux itérateurs.

24.2 Exercices

1. Écrire une classe qui permette de représenter un tableau circulaire, c'est à dire que lorsqu'il est parcouru jusqu'à la fin, on revienne au début. Par exemple, si on prend la séquence (1, 2, 3), alors cinq appels à `next()` donneront successivement : 1, 2, 3, 1, 2.

CHAPITRE 25

Persistance

La persistance réfère à l'ensemble des outils permettant de sauvegarder et réutiliser des données ou un état du programme.

25.1 Base de données

Nous étudierons le cas de la base de données SQLite3, car il s'agit de la plus simple à mettre en place [1]. Toutefois, ce qui est décrit ici fonctionnera pour toutes les bases de données, comme Python fournit une interface identique pour toutes.

Pour comprendre totalement la section qui suit, il est nécessaire d'avoir des notions sur les bases de données et le langage SQL.

La première chose à faire est de créer un objet `Connection` qui représente une base de données :

```
>>> db = sqlite3.connect('database.sqlite3')
```

Note : Il est possible d'utiliser l'argument spécial `':memory:'` afin de créer une base de données virtuelles, qui sera détruite une fois fermée. Cette technique peut être utile en vue de tester.

Afin d'exécuter une requête, il faut créer un objet `Cursor` et utiliser a méthode `execute()` :

```
>>> curs = db.cursor()
>>> curs.execute('create table persons (name, age)')
>>> curs.execute('''insert into persons values('Jean', '21')''')
>>> db.commit()
>>> curs.close()
```

Il est important de valider les requêtes avec la méthode `commit()` puis de fermer le curseur afin de ne pas gaspiller de la mémoire.

Il peut être dangereux d'insérer directement la valuer d'une variable dans une requête. À la place, il faut utiliser le symbole `?` et fournir les variables en arguments supplémentaires à `execure()` :

1. Une base de données SQLite consiste en un simple fichier sur le disque dur, au contraire de la plupart des autres bases de données qui font appel à tout un attirail pour fonctionner.

```
>>> curs = db.cursor()
>>> name = 'Jean'
>>> param = (name,)
>>> curs.execute('select * from persons where name=?', param)
>>> params = ('Michel', '10')
>>> curs.execute('insert into persons values(?, ?)', params)
>>> db.commit()
```

Après une requête `select`, il est possible d'accéder aux données en utilisant le curseur comme un itérateur, ou en utilisant les méthodes `Cursor.fetchone()` ou `Cursor.fetchall()`.

```
>>> curs.execute('select * from persons')
>>> for row in curs:
...     print(row)
('Jean', '21')
('Michel', '10')
```

Voir aussi :

http ://www.sqlite.org Le site officiel de SQLite : la documentation sur ce dernier décrit la syntaxe et les types disponibles.

PEP 249 — Database API Specification 2.0

25.2 Sérialisation

La sérialisation consiste à stocker n'importe quel objet dans un fichier, ce qui permet de le réutiliser ultérieurement, même si le programme a été interrompu entre temps. Généralement, le stockage sera effectué en binaire.

25.2.1 Pickle

Le module `pickle` permet de transformer des objets en bytes. De fait, lorsque qu'un fichier sera requis, ce dernier devra être ouvert en mode binaire.

`pickle.`**`dump`**`(`*obj*, *file*`)`
Sérialise `obj` et écrit sa représentation dans le fichier `file`.

`pickle.`**`load`**`(`*file*`)`
Lit une représentation sérialisée d'un objet dans le fichier `file` et retourne l'objet original.

Deux classes permettent de désigner une fois pour toute le fichier considéré : `Pickler(file)` (qui possède une méthode `dump()`) et `Unpickler(file)` (qui possède une méthode `load()`).

Warning : Tous les objets ne peuvent pas être sérialisés.

Il est possible de modifier le comportement d'une classe lorsqu'elle est (dé)sérialisée grâce à plusieurs méthodes :
– `__getnewargs__()` qui définit quels arguments seront passés à `__new__()`.
– Les méthodes `__getstate__()` et `__setstate__()` permettent de modifier la manière dont le dictionnaire `__dict__` est géré.
Il convient de noter que la méthode `__init__()` n'est généralement pas appelée lorsqu'un objet est désérialisé.

25.2.2 Shelve

À la différence de `pickle`, le module `shelve` permet de sérialiser plusieurs objets dans un même fichier : ces derniers sont accessibles via un système de clés, comme pour un dictionnaire [2].

`shelve.`**`open`**(*filename*)
Ouvre un dictionnaire persistant dans lequel sera stocké les données. Ce dernier possède les mêmes méthodes que les dictionnaires `dict`.

`Shelf.`**`sync`**`()`
Synchronise tous les changements mis en cache. Il est nécessaire d'appeler cette méthode pour que les changements soient écrits dans le fichier.

`Shelf.`**`close`**`()`
Cette méthode appelle `sync()` avant de fermer le dictionnaire.

2. "Shelf" veut d'ailleurs dire "étagère" en anglais.

Internationalisation et localisation

L'internationalisation et la localisation (ou régionalisation), désignées respectivement par les sigles i18n et l10n [1], consistent à :
– le premier désigne toutes les techniques permettant au logiciel de s'adapter à la culture et à la langue de l'utilisateur ;
– le second concerne plus précisément la traduction des messages affichés.
Pour ce faire, le module `gettext` fournit de nombreux outils utiles.

Ce dernier utilise des fichiers textes portant l'extension `.po` et leur version compilée `.mo` qui contiennent les traductions [2]. Usuellement, ces fichiers sont placés dans le dossier `/locale/lang/LC_MESSAGES/`, où `lang` est remplacé par le sigle de la langue voulue (`fr` pour français, `en` pour anglais...).

`gettext.`**`bindtextdomain`**(*domain*[, *localedir*])
: Associe `domain` au dossier `localedir` contenant les fichiers de localisation. Cela signifie que `gettext` va rechercher les fichiers `.mo` correspondant au domaine indiqué.

`gettext.`**`textdomain`**([*domain*])
: Si `domain` vaut `None`, alors le domaine courant est retourné, sinon, le domaine est changé à `domain`.

`gettext.`**`gettext`**(*message*)
: Retourne la traduction de `message`, basée sur le domaine actuel, le langage et le dossier de translations [3].

`gettext.`**`gettext`**(*message_sing*, *message_plur*, *n*)
: Cette fonction est utilisée pour les formes plurielles. Si une traduction trouvée, elle applique la formule à `n`, sinon elle retourne `message_sing` si `n` vaut `1` et `message_plur` sinon.

Voici un exemple :

```
>>> import gettext
>>> gettext.bindtextdomain('app', '/path/to/my/language/directory')
>>> gettext.textdomain('app')
'app'
>>> _ = gettext.gettext
>>> print _('Cette chaine est traductible.')
```

La fonction `install()` permet d'automatiser plusieurs méchanismes. Elle prend en argument le domaine et le dossier de traduction, et se charge d'ajouter l'alias `_()`.

Si certaines sont construites à partir de variables, il est conseillé d'utiliser des arguments nommés afin de permettre au traducteur de savoir qu'est-ce qui sera inséré et, surtout, pour qu'il puisse intervertir l'ordre sans souci :

1. Le chiffre indique le nombre de lettres entre les lettre initiale et finale.
2. On parle de "locale" en anglais.
3. Généralement, on définit l'alias `_()` pour la fonction `gettext()` : `` _ = gettext.gettext ``

```
>>> name = 'Anais'
>>> age = 23
>>> print _('Hello %(name)s, today is %(age)d') % {'name': name, 'age': age}
```

Toutefois, l'écriture à la main des fichiers .po est fastidieuse[4]. Pour cette raison, un logiciel permettant de simplifier grandement le travail a été conçu : poEdit. Ce dernier peut étudier le code Python afin de sélectionner les chaines gettext. Son interface permet ensuite de les traduire aisément.

Voir aussi :

http ://filyb.info/post/2006/11/02/413-tutoriel-traduire-un-programme-en-python-et-glade Cet article détaille aussi la manière d'utiliser les outils gettext sans passer par poEdit.

4. Il serait possible de récupérer les chaines et compiler les fichiers en ligne de commande, mais cette méthode ne sera pas présentée ici.

Cinquième partie

Annexes

CHAPITRE 27

Histoire

CHAPITRE 28

Évolutions

Ce chapitre liste les différences entre les versions 2.5 et 3 de Python, ce qui permettra d'adapter en conséquences le code si la version 3 n'était pas utilisée.

Bonnes et mauvaises pratiques

29.1 Imports

Il est fortement déconseillé d'utiliser l'instruction `form module import *`, car cela surcharge l'espace de noms global et peut conduire à des conflits. Observons le code suivant :

```
>>> from math import *
>>> cos(1)
0.54030230586813977
>>> from cmath import *
>>> cos(1)
(0.54030230586813977-0j)
```

L'on remarque que la fonction `cos()` du module `cmath` remplace celle du module `math`.

La seule exception notable à cette pratique est lorsque l'on utilise Python en mode interactif (par exemple en important tout le module `math` pour se servir du terminal comme d'une calculatrice).

29.2 Exceptions

Lorsqu'une erreur est interceptée, il est vital de spécifier après l'instruction `except` quelle erreur est attendue. Le cas échéant, toutes les erreurs levées seront interceptées, et cela pourra cacher certains problèmes au programmeur, et rendre le débugage très difficile. Par exemple :

```
try:
    foo = opne('file')
except:
    sys.exit('could not open file')
```

Ce code interceptera toutes les erreurs, même celle indiquant que la fonctionopn `opne()` n'existe pas, et le programmeur sera dans l'incapacité de comprendre où est le problème. Au contraire, le code suivant lui permettra de rapidement trouver l'erreur :

```
try:
    foo = opne('file')
except IOError:
    sys.exit('could not open file')
```

29.3 Utilisation des backslash

Comme il a été vu au début, un backslash permet d'écrire une expression ou une instruction sur plusieurs lignes. Toutefois, il peut être plus simple d'utiliser des parenthèses afin d'améliorer la lecture.

Les deux expressions suivantes sont équivalentes :

```
value = foo.bar()['first'][0]*baz.quux(1, 2)[5:9] \
    + calculate_number(10, 20)*forbulate(500, 360)

value = (foo.bar()['first'][0]*baz.quux(1, 2)[5:9]
    + calculate_number(10, 20)*forbulate(500, 360))
```

29.4 Conventions et style

Voici une liste (non exhaustive) des conventions communément adoptées :
– utiliser quatre espaces par niveau d'indentation, et surtout pas de tabulations :
– couper chaque ligne à 79 caractères [1] ;
– insérer une ligne vierge entre les définitions de fonctions et, classes, ainsi qu'entre les grandes articulations du code ;
– utiliser des docstrings ;
– mettre un espace de chaque côté des opérateurs, après une virgule ou deux points (mais pas avant), enfin, ne pas mettre d'espace après un signe ouvrant ou avant un signe fermant (c'est à dire `()`, `{}`, `[]`);
– nommer les classes en `CamelCase`, les fonctions, méthodes et variables en `lower_case`.
Voir aussi :

PEP 8 Style Guide for Python Code

1. Ceci pour faciliter la lecture du code et pour qu'il soit adapté à toutes les résolutions

CHAPITRE 30

Éditeurs

Il est important de choisir des outils avec lequels on se sent à l'aise, car ce sont avec ces derniers que l'on passera la majeure partie de notre temps de programmeur ! De nombreux logiciels permettent de coder, même le bloc-notes de Windows pourrait suffire ! Il est toutefois vivement conseillé de choisir un logiciel plus avancés, la coloration syntaxique étant un minimum vital.

Il ne faut pas hésiter à tester de nombreux éditeurs, à regarder sur internet comment les configurer, quels sont les avis des utilisateurs...

Dans cette annexe, nous nous concentrerons sur la présentation des interpréteurs et des éditeurs.

30.1 Interpréteurs interactifs

30.1.1 IDLE

IDLE (pour Integrated DeveLopment Environment) est un interpréteur interactif plus évolué (grâce à sa coloration syntaxique par exemple). Il s'agit d'une manière plus agréable pour débuter que le terminal Windows, qui est très austère et peu aisé d'utilisation.

Il est installé en même temps que Python sous Windows, mais il peut être nécessaire de l'installer depuis les dépôts sous Linux ou les autres systèmes, même si son intérêt est alors moindre.

30.1.2 IPython

IPython est un interpréteur interactif qui améliore significativement l'interpréteur par défaut, en proposant de nombreuses fonctionnalités :
– auto-complétion avec `tab` (comme dans le terminal Unix), prenant en compte les méthodes et attributs des objets ;
– un historique persistant des commandes exécuées ;
– une indentation automatique ;
– des raccourcis et des commandes « magiques » permettant de tester et débugger ;
– la possibilité de lancer des commandes bash ;
– sortie colorée et plus agréable.

Warning : Actuellement, iPython n'est compatible qu'avec les versions 2.x.

Je me contenterai de présenter les principales fonctionnalités et renvoie à la documentation pour un usage plus détaillé. Il est possible d'accéder à une documentation sommaire dans le terminal en tapant `?`. L'auto-complétion est très pratique à la fois éviter d'écrire complètement les commandes, mais aussi pour avoir un résumé rapide des méthodes et des attributs des objets, sans avoir besoin de consulter la documentation !

Une commande bash s'exécute en la préfixant par un point d'exclamation `!` :

```
# affichage de la date
In [1]: !date
dim. mai 30 18:23:34 EEST 2010

# création d'un dossier appelé "test"
In [2]: !mkdir test
```

Une commande magique commence par le signe pourcentage `%` (utiliser `%magic` pour accéder à la liste complète des commandes magiques ainsi qu'à leur description, et `%lsmagic` pour obtenir uniquement la liste). Les plus utiles sont :
– `%run module` : exécute un module et importe dans l'interpréteur les variables définies dans ce dernier. L'avantage est qu'à chaque `%run`, les définitions des objets sont mises à jour, contrairement à ce qui se passe lorsque avec `import`.
– `%timeit fonction` : exécute un grand nombre de fois la fonction passée en argument et affiche le temps moyen d'exécution.
De plus, l'option `-pylab` permet d'ajouter un support interactif de la librairie matplotlib, ce qui permet de tracer et analyser rapidement des données.

30.2 Environnements de développement

30.2.1 Éditeurs texte

Malgré l'apparence spartiate des éditeurs de texte, nombre d'entre eux sont largement suffisants pour le débutant, et certains peuvent même être fortement personnalisés jusqu'à devenir de vrais *IDE* ! Voici une liste des éditeurs texte les plus répandus :
– Windows : Notepad++.
– Linux : Gedit (Gnome), Kate (KDE).
– UNIX : Emacs, Vi(m) [1].

30.2.2 Eclipse (Pydev)

L'*IDE* Eclipse, à l'origine prévu pour la programmation en Java, possède un système de plugins qui permet de l'étendre si facilement qu'un grand nombre d'extensions ont vu le jour et ajoutent le support de nombreux langages et d'outils.

Le plugin dédié au Python s'appelle PyDev.

On peut noter que la suite Aptana (basée sur Eclipse, et que l'on peut ajouter en plugin à ce dernier), qui se concentre sur le développement web, contient aussi PyDev.

1. Ces deux logiciels, correctement configurés, peuvent devenir très puissants et sont d'ailleurs très utilisés par les "geeks" (qui se livrent une guerre acharnée pour déterminer lequel est le meilleur des deux). Toutefois il faut de la patience et de la pratique pour s'y habituer.

CHAPITRE 31

Outils

31.1 Générateurs de documentation

31.1.1 Sphinx

Sphinx est un logiciel qui permet de créer très facilement de belles documentations pour les logiciels Python. Toutefois, sa puissance ne le limite pas à du code Python et il peut être utilisé pour rédiger d'autres documents [1]. De nombreux formats sont disponibles pour la sortie : PDF (via LaTeX), HTML...

Le langage de balisage ReST est utilisé par Sphinx afin de mettre en forme le contenu de la documentation. Il s'agit d'un langage très simple à apprendre et intuitif.

Warning : Sphinx ne supporte pas encore la version 3 de Python.

31.1.2 Autres

D'autres outils permettant de générer des documentations existent, mais ils sont généralement moins adaptés à Python et leur rendu est moins agréable. Voici une liste non exhaustive :
– Doxygen : L'avantage de ce logiciel est de fonctionner avec la plupart des langages couramment utilisés (Python, PHP, C, Java...).
– Epydoc : Logiciel utilisé historiquement pour les documentations Python.

31.2 Pylint

Pylint est un logiciel utilisable en ligne de commande qui permet d'analyser du code, dans le but de chercher des erreurs et de vérifier la bonne rédaction (respect de la PEP 8 — voir l'annexe *Bonnes et mauvaises pratiques*).

Il peut s'utiliser pour vérifier la syntaxe d'un fichier seul ou d'un package entier. Il s'utilise comme suit :

```
$ pylint package
$ pylint file.py
```

1. D'ailleurs ce manuel a été rédigé grâce à Sphinx.

Un rapport détaillé des erreurs et des fautes de style (chacune identifiée par une catégorie) pour chaque fichier sera affiché. À la fin, le rapport indique un récapitulatif par modules et un autre global, en plus d'attribuer une note sur 10.

Pylint est très strict quant à la vérification du code, et il est souvent nécessaire de créer sa propre configuration afin qu'il accepte certaines formes qu'il considère comme des erreurs.

CHAPITRE 32

Liens

32.1 Autres tutoriels

Python Challenge http ://www.pythonchallenge.com/ : il s'agit d'un site constitué d'un ensemble d'épreuves, qui nécessitent de programmer pour être résolues.

32.2 Projets liés

Voici une liste des projets liés à Python que j'ai pu citer.

GTK http ://www.gtk.org/

Sphinx http ://sphinx.pocoo.org/

CHAPITRE 33

Glossaire

Algorithme Plan détaillant la manière de résoudre un problème.

Bytecode Code binaire intermédiaire entre le code compilé et le code source.

Code Texte formant un programme.

Débugging Détection et réparation des erreurs d'un programme.

Déclaration Instruction reliant une expression (un objet) à une variable.

Exécutable Fichier permettant de lancer le programme.

Expression Suite de symboles interprétée retournant un nouvel objet.

IDE (Integrated Development Environment) La traduction française signifie "environnement de développement intégré". Un tel logiciel a pour vocation de rassembler de nombreux outils en une seule interface.

Instruction Action à effectuer par l'ordinateur. Elle tient généralement sur une ligne.

PEP Une PEP (Python Enhancement Proposals) est un document visant à apporter des informations à la communauté ou à décrire une nouvelle fonctionnalité.

Portabilité Capacité d'un programme à fonctionner plus ou moins bien dans différents environnements.

Prédicat Un prédicat est une expression qui possède une valeur de vérité

Procédure Il s'agit d'une fonction qui ne retourne aucune valeur.

Récursivité Une fonction qui s'appelle elle-même est dite récursive.

Variable Nom d'un objet.

Vérification Démonstration qu'un programme fonctionne correctement.

CHAPITRE 34

Bibliographie

– David Ascher, Alex Martelli. *Python Cookbook*.
O'Reilly, 2005.
– Allen Downey. *Think Python : An Introduction to Software Design*.
Cambridge University Press, June 2008.
http ://www.greenteapress.com/thinkpython/.
– Mark Pilgrim. *Dive into Python 3*.
http ://diveintopython3.ep.io/.
– Gérard Swinnen. *Apprendre à programmer avec Python*.
O'Reilly, May 2005.
http ://www.cifen.ulg.ac.be/inforef/swi/python.htm.
– Wesley J. Chun. *Au coeur de Python, version 2.5, Volume 1 : Notions fondamentales*
CampusPress, 2007.

Sixième partie

Sur le livre

CHAPITRE 35

Changements

Tables des matières

35.1 Version 1.0 (en cours de rédaction)

– Partie “Introduction” : notions générales en programmation.
– Partie “Langage” : description générale du langage Python.
– Partie “Aspects spécifiques”.

CHAPITRE 36

Licence et auteurs

36.1 Licence

Ce livre est publié sous la licence Creative Commons-BY-NC-SA :

BY [Paternité] Vous devez citer le nom de l'auteur original

NC [Pas d'Utilisation Commerciale] Vous n'avez pas le droit d'utiliser cette création à des fins commerciales.

SA [Partage des Conditions Initiales à l'Identique] Si vous modifiez, transformez ou adaptez cette création, vous n'avez le droit de distribuer la création qui en résulte que sous un contrat identique à celui-ci.

Voir aussi :

Creative Commons-BY-NC-SA http ://creativecommons.org/licenses/by-nc-sa/3.0/deed.fr

36.2 Auteurs

– Harold Erbin <harold.erbin@gmail.com>

Septième partie

Indices and tables

– *genindex*
– *modindex*
– *search*
– *Glossaire*

Python Module Index

Index

Symbols

A

B

C

D

E

F

G

S

T

U

V

W