Python pour Programmeur

Cette introduction s'adresse aux programmeurs qui connaissent déjà d'autres langages de programmation procéduraux / objets, et qui désirent découvrir rapidement le langage Python. Elle n'explique donc pas les concepts de base de la programmation et est organisée dune façon totalement inadaptés à des débutants qui rechercheraient une introduction à ce domaine[1]. Le document s'attache aux spécificités de Python qui ont un impact sur la façon de programmer, peuvent être déroutantes lorsque l'on vient d'autres langages, et sont utiles à connaître pour programmer efficacement et agréablement dans ce langage.

ébauche non terminée !

29/05/13 à 21:54:52

1 Pour ceux-là, voyez les ouvrages de G. Swinnen ou de R. Cordeau, ainsi que les nombreuses ressources disponibles sur l''Internet.

Sommaire

I - DÉMARRAGE

1 - Ce document

1.a - Version de Python

Ce document se base sur **Python3** qui a été une importante évolution par rapport à Python2, tendant à nettoyer du langage certains restes historiques qui étaient conservés pour des raisons de compatibilité et pour lesquels des façons de faire plus propres et plus homogènes ont été mises en place. Python2 continue toutefois d'exister, avec la version 2.7 actuellement, mais sans évolution du langage (il y a des corrections de bogues et des adaptations pour faciliter l'écriture de code compatible entre les deux versions). Beaucoup d'informations contenues ici peuvent aussi s'appliquer à Python2, mais pas toutes. L'annexe *Python2 vs Python3*, page 55, liste une partie de ces différences.

Il existe diverses implémentations de Python, fonctionnant sur la machine virtuelle Java, sur la machine virtuelle .Net, ou encore utilisant un compilateur à la volée comme pypy. Ce document considère la version de référence de Python, « cpython », qui est de loin la plus répandue (c'est celle que l'on télécharge par défaut sur http://www.python.org/).

La sous-version actuelle, utilisée pour ce document, est **Python 3.3** qui est sorti en septembre 2012.

1.b - Styles utilisés

Les commandes lancées directement dans le shell (typiquement le lancement de l'exécution d'un fichier source Python par l'interpréteur), ainsi que leur résultat d'exécution, sont illustrées ainsi :

```
$ python3 -c "print('Un essai pour voir')"
Un essai pour voir
```

L'utilisation du shell **python3** interactif est illustrée ainsi :

```
>>> print("Tests shell Python")
Tests shell Python
>>> a = 4
>>> a+5
9
```

On utilise parfois plusieurs colonnes, lorsque l'illustration le permet.

```
>>> 2 * 5
10
>>> (4 + 1) * 12
60
```

```
>>> 1.4 / 0.5
2.8
>>> "Hello".upper()
'HELLO'
```

```
>>> "Un mot".find("mo")
3
>>> "Langage Python"[8:]
'Python'
```

Certains exemples avec le shell utilisent directement les fonctions mathématiques, considérant qu'elles ont déjà été importées du module **math**.

Enfin les fichiers script sont représentés de la façon suivante (en ayant le nom du fichier en commentaire au début de celui-ci) :

```
# file: justepourvoir.py
print("Juste pour voir !")
```

2 - Tester Sereinement

Python est très largement disponible sur les systèmes informatiques, petits et gros, toutes déclinaisons de système d'exploitation confondues. Il est souvent déjà installé, parfois même en

plusieurs exemplaires, parfois caché dans une application. Pour tester, il est possible d'utiliser un Python déjà installée… si sa version est la bonne. On peut aussi installer la version de Python qui nous intéresse. Un outil très répandu, Virtualenv (voir page 10), permet par ailleurs de mettre en place des environnements de développements parallèles complètement contrôlés et sans risque pour les versions de Python déjà installées.

2.a - Installation

Plusieurs versions de Python peuvent cohabiter sur le même système, il faut juste sélectionner le binaire à utiliser.

Sous Unix, sauf exception[2], il vaut mieux conserver une version Python 2 derrière la commande **`python`**, et il est fortement conseillé de laisser par défaut la version qui est installée en standard par/pour le système d'exploitation[3]. De même, si votre système d'exploitation package un Python 3.2 derrière la commande **`python3`**, alors il vaut mieux le laisser et installer un Python plus récent en parallèle, qu'on lancera simplement en précisant son numéro de version, **`python3.3`** par exemple, ou en spécifiant un répertoire d'installation personnalisé **`~/altroot/bin/python3`** par exemple. L'installation d'une version plus récente peut se faire via le système de paquets de la distribution utilisée si cette version est disponible dans les dépôts, ou sinon en compilant le programme à partir des sources comme indiqué ci-dessous.

Sous Windows, chaque installation de Python peut se faire dans un répertoire au choix — par défaut **`C:\Python<version>`**. Et chacune installe des entrées spécifiques dans le menu **`Démarrer`** ⟹ **`Programmes`**.

La dernière version de Python peut être récupérée directement sur la page de téléchargement Python http://www.python.org/getit/, où l'on trouve des archives directement installables pour certains systèmes d'exploitation (Windows[4] ou les versions récentes de MacOSX), ainsi que les sources que l'on peut compiler.

Compilation sous Unix

La compilation sous Linux/Unix, ne pose pas spécialement de problème, le plus difficile étant parfois de disposer des versions de développement des différentes librairies utilisées par Python (readline pour la gestion de la ligne d'édition, tcl/tk pour l'interface graphique standard, etc — certaines sont optionnelles et les services correspondant sont indisponibles si les librairies ne sont pas trouvées lors de la construction). Il est fortement conseillé de spécifier un répertoire d'installation hors de l'arborescence standard des commandes du système, afin de ne pas interférer avec celles-ci.

Des recettes de compilation se trouvent sur le net, par exemple pour Python 3.3 avec Ubuntu 12.04 LTS (et ses dérivées) http://askubuntu.com/questions/244544/how-to-install-python-3-3 (j'ai utilisé la solution basée sur **`apt-get build-dep python3.2`**, suivi de l'installation de librairies complémentaires puis du téléchargement et décompression des sources, et enfin de la compilation standard via **`./configure --prefix=$HOME/altroot`** puis les **`make`** et **`make install`**).

Une solution sous Unix est d'utiliser *pythonbrew*[5], qui automatise l'installation et l'utilisation de versions différentes de Python dans le répertoire utilisateur.

2 La distribution Arch Linux a choisi Python 3 comme binaire par défaut pour python.
3 Ou l'application qui l'a installé pour son usage.
4 Pour Windows, les derniers installeurs Python comportent une option pour inclure le répertoire d'installation de Python dans les chemins du `PATH` où sont recherchés les exécutables.
5 Disponible sur https://github.com/utahta/pythonbrew.

2.b - Outils

Comme beaucoup de langages de scripts, Python dispose d'un mode « shell » avec exécution immédiate des instructions et affichage des résultats. Sous Unix, il suffit de lancer **`python3`** en ligne de commande, sans rien spécifier, pour démarrer une session interactive. Si vous avez une version de Python plus personnalisée (voir *Installation*, page 8), il faut l'appeler explicitement, par exemple : **`~/altroot/bin/python3`**.

```
laurent@litchi:~$ altroot/bin/python3
Python 3.3.1 (default, May  8 2013, 19:53:25)
[GCC 4.6.3] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Sous Windows, chaque version de Python installée crée une entrée spécifique dans le menu des applications, qu'il suffit d'appeler : **`Démarrer ⟹ Programmes ⟹ Python 3.3 ⟹ Python (command line)`**.

Une fois le shell interpréteur lancé, on trouve un prompt `>>>` (avec un prompt de continuation `...` lorsqu'une expression n'est pas syntaxiquement terminée), la validation provoquant l'exécution de la ligne en cours, ainsi que l'affichage de la valeur résultante si la ligne contenait une expression (sauf si celle-ci s'évalue à la valeur **`None`**, qui n'est pas affichée).

```
>>> print("Tests shell Python")
Tests shell Python
>>> a = 4
>>> a+5
9
```

Le shell python3 de base offre quelques services de rappel et d'édition des lignes, mais il reste limité. Pour gagner (beaucoup) en fonctionnalités, on peut installer **ipython3**[6]. Il existe aussi un outil shell dans une interface graphique, standard en Python, **idle3**[7], qui permet en plus d'éditer les scripts.

2.c - Packages

Python vient « batteries included », c'est-à-dire avec un nombre important de packages[8] disponibles en standard. Parmi les nombreux autres packages, certains sont directement disponibles et installables via le *gestionnaire de paquets* (systèmes Unix) ou via un *installeur* (système Windows), d'autres doivent être téléchargées et installées spécifiquement. On peut aussi avoir besoin d'une installation spécifique pour mettre en œuvre un package avec une version différente de celle proposée par le système ou pour une version personnalisée de Python.

Le dépôt central sur lequel on trouve une grande partie des packages pour Python est le *Python Package Index*, appelé **PyPI**, sur https://pypi.python.org/pypi. On trouve aussi beaucoup de packages Python sur leurs propres pages web ou via des dépôts de gestionnaires de versions.

L'installation « à l'ancienne » consistait simplement à copier les fichiers dans les bons répertoires. Ensuite un fichier **`setup.py`** a été livré avec les packages, qui facilitait l'installation en exécutant simplement une commande **`python setup.py install`**.

---

6 IPython est disponible sur http://ipython.org/, il offre beaucoup de services (calcul, tracé de courbes, etc.). Voir aussi Spyder (disponible sur http://code.google.com/p/spyderlib/), qui est un environnement de développement scientifique autour de Python, intégrant IPython.

7 Idle3 est installé par défaut sous Windows lorsqu'on installe Python. Sous Linux il fait généralement partie d'un package particulier à installer. Et sous MacOSX, il faut télécharger un installeur spécifique adaptée à la version du système d'exploitation (voir http://www.python.org/getit/mac/tcltk/).

8 On pourrait utiliser le terme de librairies, mais pour Python on parle plutôt de packages — qui a un sens spécifique au niveau de l'organisation des fichiers.

Le fichier **setup.py** existe toujours, mais il est maintenant utilisé par les outils de gestion de packages en relation avec PyPI, permettant au créateur du package de le téléverser sur les serveurs PyPI, et aux utilisateurs de le télécharger ainsi que ses dépendances et de l'installer.

Le principal outil pour installer de façon automatique des packages Python est **pip**[9], qui est devenu l'outil de référence et s'intègre avec *virtualenv*, présenté ci-après. On dispose aussi d'un outil plus ancien, mais encore utile, **easy_install**[10].

2.d - Virtualenv

Cet outil permet de partir d'une installation existante de Python (qu'il faut donc avoir fait), et d'en créer un duplicata sur lequel on va pouvoir installer les packages que l'on désire dans les versions dont on a besoin, sans aller toucher à l'installation d'origine de Python (par exemple celle du système). Le duplicata est basé en partie sur des liens qui évitent de dupliquer complètement l'installation d'origine. On peut ainsi conserver une installation système de Python propre, qui utilise le gestionnaire de paquets pour installer les packages Python dans les versions garanties homogènes par les responsables du packaging du système d'exploitation, et N installations qui vont directement chercher les packages Python dans PyPI.

On le trouve directement sur pypi https://pypi.python.org/pypi/virtualenv.

Exemple virtualenv pyramid

Par exemple, avec la commande suivante on crée un environnement basé sur le Python 3.3 qu'on a compilé, ne contenant que les paquets standards (option **--no-site-packages**), et qui est créé dans un répertoire **envpyramid**.

```
$ virtualenv --no-site-packages -p ~/altroot/bin/python3 envpyramid
Running virtualenv with interpreter /home/laurent/altroot/bin/python3
Using base prefix '/home/laurent/altroot'
New python executable in envpyramid/bin/python3
Also creating executable in envpyramid/bin/python
Installing distribute.............   …   ..done.
Installing pip................done.
```

On obtient une arborescence qui contient **python** ainsi que **pip** et **easy_install**, et toutes les dépendances nécessaires.

```
envpyramid/
├── bin
│   ├── activate
│   ├── activate.csh
│   ├── activate.fish
│   ├── activate_this.py
│   ├── easy_install
│   ├── easy_install-3.3
│   ├── pip
│   ├── pip-3.3
│   ├── python -> python3
│   ├── python3
│   └── python3.3 -> python3
├── include
│   └── python3.3m -> /home/laurent/altroot/include/python3.3m
└── lib
    └── python3.3
        ├── abc.py -> /home/laurent/altroot/lib/python3.3/abc.py
        …
```

9 Voir sur http://www.pip-installer.org/en/latest/.

10 Voir sur http://pythonhosted.org/distribute/easy_install.html.

Pour utiliser cet environnement Python, on peut explicitement utiliser les binaires **`envpyramid/bin/python`**, **`envpyramid/bin/pip`**, etc. Mais on peut aussi "sourcer" le script d'activation de l'environnement :

```
laurent@litchi:~/venv$ cd envpyramid/
laurent@litchi:~/venv/envpyramid$ . bin/activate
(envpyramid)laurent@litchi:~/venv/envpyramid$
```

Le prompt est changé pour indiquer qu'on utilise l'environnement choisi (le PATH est modifié de façon à avoir le chemin absolu de **`envpyramid/bin`** en tête). Les commandes **`python`**, **`pip`**, **`easy_install`** sont donc celles de l'environnement et agissent sur celui-ci.

Ensuite, installation des packages désirés, avec leurs dépendances.

```
(envpyramid)laurent@litchi:~/venv/envpyramid$ pip install pyramid
Downloading/unpacking pyramid
  Downloading pyramid-1.4.1.tar.gz (2.4MB): 2.4MB downloaded
  Running setup.py egg_info for package pyramid

Requirement already satisfied (use --upgrade to upgrade): distribute in
./lib/python3.3/site-packages/distribute-0.6.34-py3.3.egg (from pyramid)
Downloading/unpacking Chameleon>=1.2.3 (from pyramid)
  Downloading Chameleon-2.11.tar.gz (157kB): 157kB downloaded
  Running setup.py egg_info for package Chameleon

… <zip> l'installation est verbeuse et ramène beaucoup de dépendances …

Successfully installed pyramid Chameleon Mako WebOb repoze.lru zope.interface
zope.deprecation venusian translationstring PasteDeploy MarkupSafe
Cleaning up...
```

On peut maintenant utiliser l'environnement virtuel pour développer et tester.

```
(envpyramid)laurent@litchi:~/venv/envpyramid$ python
Python 3.3.1 (default, May  8 2013, 19:53:25)
[GCC 4.6.3] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pyramid
>>>
```

Virtualenvwrapper

Il s'agit d'un ensemble d'extensions à virtualenv, qui facilitent la gestion des environnements virtuels créés. Il fournit plusieurs commandes en plus, avec le support de la complétion et en installe des scripts « hook » qui permettent d'automatiser certaines opérations lors des manipulations des environnements virtuels.

On le trouve sur https://bitbucket.org/dhellmann/virtualenvwrapper/. Voir sa documentation pour l'installation (entre autre la variable d'environnement **`WORKON_HOME`**). Un petit exemple :

```
laurent@litchi:~$ workon envpyramid
(envpyramid)laurent@litchi:~$ lssitepackages
chameleon                           repoze
Chameleon-2.11-py3.3.egg-info       repoze.lru-0.6-py3.3.egg-info
distribute-0.6.34-py3.3.egg         repoze.lru-0.6-py3.3-nspkg.pth
easy-install.pth                    setuptools.pth

… <zip> pyramid a beaucoup de dépendances …

pyramid                             zope.interface-4.0.5-py3.3.egg-info
pyramid-1.4.1-py3.3.egg-info        zope.interface-4.0.5-py3.3-nspkg.pth
pyramid_debugtoolbar-1.0.6-py3.3.egg
```

II - LES BASES

On présente ici les bases sur l'écosystème Python, qui permettent ensuite d'aborder certains outils et certaines recettes typiques du langage en comprenant pourquoi il faut utiliser cela ou faire ainsi.

1 - Évolutions du langage

Le langage est issu des travaux de Guido van ROSSUM (GVR) qui a démarré le développement de Python en décembre 1989, la version 1.0 de Python étant sortie en janvier 1994. Le développement de Python se fait avec des contributions et discussions de toute une communauté qui s'est formée autour du langage, mais est toujours guidé par GVR qui en est le « dictateur bénévole à vie » (BDFL - Benevolent Dictator For Life) et à qui reviennent les décisions finales.

Depuis mars 2001, une association à but non lucratif, la *Python Software Foundation* (PSF) est en charge de diverses taches dans la communauté Python, entre autres le développement du langage, la gestion des droits autour du langage, l'organisation de conférences de développeurs (les PyCon), la levée de fonds pour fonctionner. Il n'y a donc pas, derrière Python, de grosse structure commerciale ou académique, mais une forte communauté.

Au niveau des évolutions de Python, une organisation un peu similaire aux RFC[11] de l'IETF[12] a été mise en place, les **Python Enhancement Proposals** (PEP). Ce sont des documents qui formalisent des propositions concernant le langage ou encore les librairies standard, et qui sont discutés, amendés - sous la responsabilité du rédacteur du PEP qui doit arriver à un consensus - pour être finalement acceptés ou refusés.

Voir les PEPs par ici : http://www.python.org/dev/peps/

Les évolutions du langage essaient de conserver à celui-ci sa lisibilité et son efficacité, reprenant des fonctionnalités d'autres langages lorsqu'elles apparaissent pertinentes.

1.a - Le Zen de Python

Les principes qui guident la conception de Python, et que les utilisateurs essaient de retrouver en codant dans ce langage, ont été exprimés par Tim PETERS dans le PEP n°20[13] (elles sont accessibles dans le shell Python en exécutant `import this`). En voici une version traduite par Cécile TREVIAN et Bob CORDEAU.

Préfère
la beauté à la laideur,
l'explicite à l'implicite,
le simple au complexe,
le complexe au compliqué,
le déroulé à l'imbriqué,
l'aéré au compact.
Prends en compte la lisibilité.
Les cas particuliers ne le sont jamais assez pour violer les règles.
Mais, à la pureté, privilégie l'aspect pratique.
Ne passe pas les erreurs sous silence,

11 RFC : Request For Comments
12 IETF : Internet Engineering Task Force
13 PEP n°20 « The Zen of Python » : http://www.python.org/dev/peps/pep-0020/

Ou bâillonne-les explicitement.
Face à l'ambiguïté, à deviner ne te laisse pas aller.
Sache qu'il ne devrait y avoir qu'une et une seule façon de procéder.
Même si, de prime abord, elle n'est pas évidente, à moins d'être Néerlandais.
Mieux vaut maintenant que jamais.
Cependant jamais est souvent mieux qu'immédiatement.
Si l'implémentation s'explique difficilement, c'est une mauvaise idée.
Si l'implémentation s'explique aisément, c'est peut-être une bonne idée.
Les espaces de noms, sacrée bonne idée ! Faisons plus de trucs comme ça!

2 - Modèle d'exécution

L'implémentation standard de Python (« cpython »), est basée sur une machine virtuelle codée en C qui interprète un code objet dédié (bytecode). Le bytecode résulte de la compilation des modules sources Python (fichiers `.py`[14]).

Toutefois, une grande partie du cœur de Python et des librairies annexes est écrite en langage C et compilée en code natif (fichiers `.so` sous Linux, `.dll` sous Windows). Une interface de programmation normalisée permet l'utilisation transparente à l'exécution de code natif et de bytecode, qui peuvent s'appeler réciproquement.

Le mélange de code interprété et compilé permet d'obtenir de bonnes performances pour autant qu'on réussisse à effectuer les calculs lourds et répétitifs dans le code compilé - de nombreuses librairies scientifiques offrent les services d'algorithmes compilés en natif. La façon dont l'interface de programmation est construite permet un remplacement, sans modification pour l'utilisateur, d'un module de code Python et son bytecode interprété par un module de code en C compilé ; il est ainsi possible de développer rapidement une version de test en Python, puis de passer en C uniquement les modules dont le coût en temps de calcul le justifie.

2.a - Tout objet

Le fonctionnement interne de Python repose sur un modèle où tout est objet, non seulement les données manipulées, types de base ou conteneurs, fichiers, mais aussi tous les objets manipulés par le langage : classes, fonctions et méthodes, code, modules et packages, exceptions, pile d'exécution…. Tout.

Cela n'empêche pas de permettre une programmation procédurale simple, où l'on définit des fonctions et où on utilise de façon transparente des données de types prédéfinis.

2.b - Tout typé

Python est un langage où *toutes les valeurs sont typées*, toutes relèvent d'une classe qui définit les opérations que l'on peut faire dessus ainsi que les combinaisons possibles avec d'autres types. Par contre, *les variables sont dynamiquement typées* : il est possible de réaffecter une variable avec une valeur d'un type différent (voir *Affectation* page 23).

Note : dans ce document on utilise indifféremment les termes *type* ou *classe*. Pour Python il s'agit de la même chose, car les types et classes sont unifiés dans un système commun.

2.c - Duck typing

Python utilise le « *Duck Typing* », que l'on peut traduire en typage d'interface : si on attend d'un objet canard qu'il fasse coincoin et comporte des pattes palmées, alors tout objet fai-

14 On trouve aussi des `.pyw` sous Windows, qui évitent l''ouverture de la console lors de l''exécution.

sant coincoin et ayant des pattes palmées sera considéré comme un canard et utilisé en tant que tel.

Il n'y a pas de définition de contrat et de contrôle a priori, c'est à l'exécution qu'il sera vérifié qu'une valeur manipulée supporte bien les opérations et attributs que l'on attend d'elle. Des outils comme les « Abstract Base Class » permettent de mettre en place de tels contrôles à l'exécution si on le juge nécessaire (voir *Classes abstraites*, page 52).

2.d - Gestion mémoire

En Python (cpython), la gestion de la mémoire est basée sur un comptage de références par objet, toute nouvelle référence sur un objet incrémente ce compteur, toute suppression de référence le décrémente. Quand le compteur passe à zéro l'objet est supprimé.

Afin de pouvoir libérer les objets devenus inutiles, mais qui restent référencés en raison de références circulaires, un ramasse-miettes est utilisé périodiquement. Il peut être désactivé (si l'on est sûr de correctement gérer les références circulaires) et réactivé dynamiquement ; les fonctions qui permettent de le contrôler sont dans le module standard **`gc`**.

3 - Modules

3.a - Nommage

Il y a une association un à un entre l'identificateur d'un module Python et le fichier sur disque correspondant à ce module, que ce soit un script Python ou un module C compilé. L'identificateur du module est simplement *le nom du fichier avant l'extension* (donc le **`.py`** exclu).

Par exemple au module **`calendar`** correspond le fichier **`calendar.py`**.

Ceci implique que **les noms des fichiers de modules sources Python doivent respecter les règles des identificateurs** (cf Identificateurs page 18) pour pouvoir être importables.

3.b - En-tête

Les deux premières lignes des modules peuvent contenir, sous forme de commentaire (ligne commençant par un `#`), des informations particulières :

- En *première ligne* obligatoirement, le *shebang*, directive débutant par `#!` au début du fichier pour indiquer aux shells qu'il s'agit d'un fichier de script, et qui se poursuit par l'indication de l'interpréteur de commandes à utiliser pour exécuter le fichier.
- En *première ou seconde ligne* (seconde si le shebang a été placé sur la première), une directive indiquant à Python dans quel encodage de caractères le fichier a été enregistré[15] — très important dès que l'on a des chaînes de caractères contenant autre-chose que de l'ASCII de base.

  L'identification de cette directive d'encodage est basée sur la recherche du modèle d'expression régulière `"coding[:=]\s*([-\w.]+)"`, ce qui permet d'identifier les indications d'encodage utilisés par divers éditeurs.

Exemple d'en-tête :

```
#!/usr/bin/env python3
# -*- encoding : utf-8 -*-
```

15 Voir le PEP263 « Defining Python Source Code Encodings » : http://www.python.org/dev/peps/pep-0263/

L'utilisation de `/usr/bin/env python3` pour indiquer l'interpréteur permet d'utiliser l'interpréteur `python3` disponible dans le `PATH` de l'utilisateur, sans savoir a priori où il a été installé.

L'encodage par défaut est l'ASCII, mais on privilégie maintenant l'utilisation explicite de l'UTF–8 qui est plus pratique pour représenter une large gamme de caractères et qui est supporté par de nombreux éditeurs de texte.

3.c - Exécution du module principal

Le module principal est celui qui est chargé en premier, au démarrage de Python. C'est le module donné en argument sur la ligne de commande, ou encore celui démarré en premier quand on demande l'exécution du script dans un environnement de développement. Il peut être identifié en utilisant la variable globale **`__name__`** définie dans et pour chaque module.

Par exemple, en définissant le module **`quisuisje`** suivant :

```
# File: quisuisje.py
print("Je suis", __name__)
```

Il est possible de lancer le module **`quisuisje`** en tant que programme principal en indiquant le fichier correspondant à Python, auquel cas la variable globale **`__name__`** de ce module vaut `"__main__"` :

```
$ python3 quisuisje.py
Je suis __main__
```

Sinon, en ayant démarré une session interactive (ou à partir d'un autre module), il est possible d'importer le module avec l'instruction **`import quisuisje`**, mais alors il n'est plus le module principal et sa variable globale **`__name__`** vaut `"quisuisje"` (l'identificateur du module) :

```
>>> import quisuisje
Je suis quisuisje
```

Cette distinction module principal ou non est couramment utilisée afin de n'exécuter certaines séquences de code que si le module est le module principal.

Dans le cas de ce que l'on appelle l'autotest, où un module outil intègre son propre code de test que l'on exécute simplement en utilisant le module comme module principal.

Dans le cas d'un module fournissant des services, mais qui est aussi utilisable de façon autonome si le module est démarré comme module principal.

3.d - Chargement à la demande

L'exécution de l'instruction **`import`** permet de demander l'accès à un module particulier. Si le module requis n'est pas déjà en mémoire, ceci entraîne son **chargement** et son **exécution** (avec si besoin la compilation préalable du code Python en bytecode).

Pour un module en Python, c'est l'*exécution du module lors de son chargement* qui provoque la mise en place de toutes les définitions qu'il peut contenir (fonctions, classes, variables). Pour un module en C une fonction spécifique d'initialisation est appelée, juste après le chargement du code objet exécutable, et est chargée de mettre en place les éléments constitutifs du module.

3.e - Chemins de recherche

Les modules sont recherchés dans le *path*[16] *Python* , simple liste de répertoires construite à partir de chemins de base référençant les répertoires d'installation des librairies Python (correspondant au binaire Python utilisé), et du contenu de la variable d'environnement `PYTHONPATH`[17].

16 Équivalent du PATH pour les binaires sous Unix et Windows.
17 Sous Windows certaines clés du registre sont aussi utilisées.

La liste de ces répertoires de recherches est accessible et modifiable à l'exécution via la variable globale **path** du module standard **sys** (variable contenant une liste de chaînes). Il est ainsi possible de modifier dynamiquement la liste des répertoires utilisés, par exemple en insérant au début de cette liste un répertoire contenant des modules personnels :

```
>>> import sys
>>> sys.path
['', '/home/laurent', '/usr/lib/python3.2', '/usr/lib/python3.2/plat-linux2',
'/usr/lib/python3.2/lib-dynload', '/usr/local/lib/python3.2/dist-packages',
'/usr/lib/python3/dist-packages']
>>> sys.path.insert(0, "/opt/unsoft")
>>> sys.path
['/opt/unsoft', '', '/home/laurent', '/usr/lib/python3.2',
'/usr/lib/python3.2/plat-linux2', '/usr/lib/python3.2/lib-dynload',
'/usr/local/lib/python3.2/dist-packages', '/usr/lib/python3/dist-packages']
```

3.f - Cache

Afin d'éviter des phases de recompilation inutiles (code source Python inchangé), le bytecode résultant de la compilation d'un module est stocké sur disque et réutilisé lorsque c'est possible. Cela donne lieu à la création de fichiers **.pyc** (ou **.pyo** si l'option d'optimisation a été activée lors du lancement de Python — voir *Options Python en ligne de commande*, page 55).

Les anciennes versions de Python stockaient ces fichiers directement à côté du fichier **.py** de code source Python correspondant. À partir de Python 3.2, un répertoire **__pycache__** est créé pour contenir ces fichiers, et un suffixe "magic" spécifique à la version de Python utilisée est ajouté aux fichiers bytecode créés dans ce répertoire. Ceci évite de polluer les répertoires de sources, et permet d'avoir plusieurs Pythons partageant le même code sans écraser mutuellement leurs fichiers bytecode qui peuvent différer suivant la version de Python.

Pour le module **calendar**, auquel correspond le fichier source **calendar.py**, il y a donc par exemple un fichier bytecode **__pycache__/calendar.cpython-32.pyc**.

3.g - Packages

Un package (paquet) est le regroupement d'un ensemble de modules dans un répertoire. Le nom du répertoire est utilisé comme nom du package lors de l'import, ce qui a les mêmes implications que pour les noms des modules : les répertoires de packages doivent respecter des règles des identificateurs.

Pour indiquer qu'un répertoire est bien un package Python, celui-ci doit de plus contenir un fichier module Python **__init__.py**, éventuellement vide. Ce module est automatiquement importé lorsque le package est importé, et permet de mettre en place des initialisations globales au package ou de définir des noms directement liés au package lui-même.

Un package peut contenir des sous-packages (sous-répertoires), qui doivent respecter les mêmes règles d'organisation et de nommage. Les fichiers **__init__.py** de chaque niveau sont chargés au fur et à mesure, en descendant dans l'arborescence des packages et sous-packages, jusqu'à atteindre le module à charger.

Soit l'arborescence suivante, dans laquelle chaque module affiche simplement lorsqu'il est importé et définit une variable spécifique :

```
├── packageniv1                                  variables définies
│   ├── __init__.py                              vp1=3
│   ├── moduleniv1.py                            vp1m1=5
│   ├── packageniv2
│   │   ├── __init__.py                          vp2=1
│   │   └── moduleniv2.py                        vp2m2=8
│   ├── packageniv2_2
```

```
│   │   ├── __init__.py                    vp22=9
│   │   ├── moduleniv2_2.py                vp22m2=12
```

Le module **moduleniv1** en exemple :

```
print("chargement packageniv1.moduleniv1")
vp1m1 = 5
```

Les imports produisent les chargement et exécution des modules nécessaire pour accéder au module désiré, en descendant dans l'arborescence.

```
>>> import packageniv1
chargement packageniv1.__init__
>>> import packageniv1.moduleniv1
chargement packageniv1.__init__
chargement packageniv1.moduleniv1
>>> import packageniv1.packageniv2
chargement packageniv1.__init__
chargement packageniv1.packageniv2.__init__
>>> import packageniv1.packageniv2.moduleniv2
chargement packageniv1.__init__
chargement packageniv1.packageniv2.__init__
chargement packageniv1.packageniv2.moduleniv2
```

Note : dans l'exemple, on a redémarré le shell python entre deux imports.

4 - Identificateurs

Les règles de formation des identificateurs sont les règles fréquemment utilisées : commencer obligatoirement par un caractère alphabétique ou un tiret souligné (_), et poursuivre éventuellement avec une série de caractères alphabétiques, chiffres et tirets soulignés. Exemples : **x**, **maxi**, **leGrosChat**, **un_mot**, **Animal**, **alpha3**, **zip4_2**, **_toto**, **__init__**.

En Python **la casse est significative**, **toto**, **Toto** et **TOTO** sont des identificateurs différents qui peuvent être utilisés dans le même code.

Les caractères alphabétiques couvrent tous les **alphabets de l'Unicode**. Les caractères accentués, les alphabets non latins (grec, kanji...) sont valides[18].

```
>>> λ=0.0123
>>> π=3.14159
```

```
>>> ℚ=λ+π
>>>
```

```
>>> ℚ
3.15389
```

Par convention, les identificateurs comportant *un trait souligné au début* (ex. **_minmax**) sont considérés comme relevant d'un usage interne et ne devraient jamais être utilisés par des tiers (leur sens et leur usage dans le code peut être amené à évoluer). La présence de ce trait souligné au début a aussi un impact sur l'import d'identificateurs globaux définis dans un module.

Les identificateurs comportant *deux traits soulignés au début* (ex. **__prenom**) ont un comportement particulier lorsqu'ils sont utilisés dans les définitions de classes. Afin d'éviter les collisions de noms dans les sous-classes, ils sont préfixés lors de la compilation en bytecode par un souligné et le nom de la classe. Par exemple l'identificateur **__prenom** utilisé comme attribut dans la définition de la classe **Bonzhomme** est transformé en **_Bonzhomme__prenom**.

Les identificateurs comportant *deux traits soulignés au début et à la fin* (ex. **__add__**) sont a priori réservés pour un usage interne par Python (un certain nombre de tels identificateurs ont déjà un usage dédié).

Il n'y a **aucune sécurité de contrôle d'accès** mise en place par le langage pour prévenir l'accès à ce qui est référencé par un identificateur (pas de notion de *private* ou *protected* ou *public*),

18 Ceci est nécessaire par exemple pour faciliter l''apprentissage de la programmation avec Python dans les écoles de pays utilisant des écritures non latines - permets aux élèves d''avoir des identificateurs significatifs pour eux —, mais il est fortement conseillé de se limiter à l''ASCII 7 bits si le code est potentiellement diffusable internationalement.

les conventions permettent d'indiquer ce qu'il faut éviter d'accéder directement, et c'est au développeur de les respecter ; mais si on connaît un nom, il y a moyen d'y accéder.

4.a - Conventions de nommage

Les règles de bons usages pour le choix des identificateurs sont formalisées dans le PEP8 « style guide for Python code ». Rapidement : `unmodule` (minuscules, pas trop long, traits soulignés si cela améliore la lisibilité), et `unpackage` (minuscules, pas trop long), `une_fonction` ou `une_methode` (minuscules avec des traits soulignés si cela améliore la lisibilité), `UneClasse` (camel-case sans trait souligné), `UneClasseError` (classe d'exception terminée par *Error*), `UNE_CONSTANTE` (majuscules avec des traits soulignés entre les mots), `une_variable_globale` ou une `variable_d_instance` (minuscules avec des traits soulignés si cela améliore la lisibilité).

Cas spécifiques : `type_` (trait souligné à la fin pour une variable reprenant un mot clé Python), `_socket` (trait souligné au début pour un module C ayant un module Python correspondant qui fournit un service de plus haut niveau - ici le module `socket`).

Le premier paramètre d'une méthode d'instance, représentant l'objet manipulé, est nommé `self`. Celui d'une méthode de classe, représentant la classe, `cls`.

Certains modules standard de Python ayant été développés avant que ces conventions ne soient établies, ils ne les respectent pas toujours.

5 - Espaces de noms dynamiques

En Python on manipule des **noms** (identificateurs), qui existent dans des hiérarchies d'**espaces de noms**, et référencent des **objets**. Ce fonctionnement est central, tout marche de cette façon :

- les fonctions de base disponibles en standard dans le langage sans ne rien avoir à importer (les « builtin ») sont accessibles dans un espace de noms `__builtins__`,
- les variables, fonctions, classes définies dans un module (Python ou natif), ou importées, sont accessibles dans un espace de **noms globaux** *du module*,
- les arguments et variables locales à l'appel d'une fonction ou d'une méthode sont accessibles dans un espace de **noms locaux** *de cet appel* de la fonction ou méthode,
- les attributs et méthodes d'une classe sont accessibles dans un espace de noms de cette classe, espace qui s'étend pour la recherche vers les espaces de noms des classes parentes,
- les attributs et méthodes d'un objet sont accessibles dans un espace de noms de cet objet, espace qui s'étend pour la recherche vers les espaces de noms des classes dont il est instance.

La fonction standard `dir()`, appelée sans paramètre, retourne la liste des noms définis dans l'espace courant de noms (noms locaux quand appelée dans une fonction/méthode, et noms globaux quand appelée dans un module ou à partir du shell). Lorsqu'elle est appelée avec comme paramètre un espace de noms particulier, elle retourne la liste des noms définis dans cet espace.

```
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__']
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
#… il y a 132 noms dans les builtins …
'vars', 'zip']
```

5.a - Import de modules / noms

Lorsque l'on exécute une instruction **import**, le premier effet est de charger le module s'il n'est pas déjà présent en mémoire (cf *Chargement à la demande* page 16), le second effet est de créer un ou plusieurs noms dans l'espace courant de noms (celui du module qui effectue l'import).

Si on utilise simplement l'*import d'un module* via l'instruction **import nommodule**, alors un nouveau nom **nommodule** est créé dans l'espace courant de noms. Ce nom est une référence directe vers le module qui vient d'être chargé, et permet d'accéder à tout ce qui est défini au niveau global de ce module - en passant par le nom du module importé.

```
>>> import math
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'math']
>>> dir(math)
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh',
'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e',
'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp',
'fsum', 'gamma', 'hypot', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma',
'log', 'log10', 'log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt',
'tan', 'tanh', 'trunc']
>>> math.pi
3.141592653589793
>>> math.sin(0.707)
0.6495557555564224
```

Si on utilise l'*import sélectif d'une liste de noms* via l'instruction **from nommodule import nom1, nom2…**, alors les noms spécifiés après import sont simplement dupliqués à partir du module importé vers l'espace courant de noms (attention, les noms sont dupliqués, mais ils référencent les mêmes objets).

```
>>> from math import pi, cos
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'cos', 'pi']
>>> pi
3.141592653589793
>>> cos(0.5)
0.8775825618903728
```

Il est possible d'utiliser les deux types d'import sur un même module : importer le module en bloc pour pouvoir accéder à certaines globales de ce module dans leur espace de nom (on verra l'importance de ceci avec l'affectation), et ensuite importer divers noms de fonctions de ce module pour pouvoir les utiliser sans avoir à passer par le nom du module.

```
>>> import sys
>>> from sys import getfilesystemencoding
>>> getfilesystemencoding()
'utf-8'
>>> sys.path
['', '/home/pointal', '/usr/lib/python3.2', '/usr/lib/python3.2/plat-linux2',
'/usr/lib/python3.2/lib-dynload', '/usr/local/lib/python3.2/dist-packages',
'/usr/lib/python3/dist-packages']
```

S'il y a beaucoup de noms à importer, il est possible de les regrouper entre des parenthèses afin d'exprimer l'instruction d'import sur plusieurs lignes.

```
from math import (acos, asin, asinh, atan2, ceil, cos, cosh,
                  degrees, radians, exp, factorial, floor, gamma)
```

Attention : lors du chargement d'un module, *les noms sont définis au moment de l'exécution de leur définition dans le module*. Un nom existant peut être *redéfini par une instruction* exécutée ultérieurement.

> Les noms définis dans le module peuvent ainsi être redéfinis par des noms importés et vice-versa, suivant l'ordre dans lequel ont été placées les instructions de définition et d'import — la dernière instruction exécutée l'emportant.

*Import **

La notation `from nommodule import *`, qui permet d'importer directement l'ensemble[19] des noms d'un module, est à éviter hors phases de tests rapides, car on ne contrôle pas ce qui est importé.

On risque l'écrasement de noms comme indiqué dans l'encadré, ou encore le masquage de noms standards par des noms identiques utilisés par le module importé. L'exemple typique est `from os import *`, qui provoque l'import d'une fonction `open` issue de ce module, ayant une signature différente de la fonction `open` standard des builtins et venant masquer celle-ci dans l'ordre de résolution des noms (cf *Résolution des noms* ci-après).

La liste des noms exportés via `*` peut toutefois être contrôlée par le module qui les définit, en créant une variable globale `__all__` contenant, la liste des symboles qui seront importés par `*` (`__all__` est simplement une liste de chaînes).

On peut aussi exclure certains noms de l'import * en les faisant commencer par un tiret souligné (_) — ça peut être utile dans un module définissant beaucoup de symboles, pour éviter d'avoir à maintenir une liste `__all__` trop importante.

Renommage

Si certains noms sont considérés comme étant trop longs, ou encore pour résoudre des collisions de noms qui provoqueraient des redéfinitions malencontreuses, il est possible de spécifier des alias pour les noms que l'on importe en utilisant l'instruction `as` lors de l'import.

```
>>> import math as m
>>> m.pi
3.141592653589793
>>> from math import cos as Cosinus, sin as Sinus
>>> Cosinus(0.1)
0.9950041652780258
```

Import de packages

Le regroupement de modules en packages et sous-packages (voir *Packages*, page 17) ajoute des niveaux hiérarchiques lors de l'import. On peut importer un package complet avec `import nompackage`, ce qui fait charger son module `__init__` et rend les noms qui y sont définis accessibles via l'espace de noms du package. Mais on peut aussi importer un sous-package ou un module du package ou d'un sous-package *en indiquant les noms de packages/sous-packages à traverser* avec la notation pointée.

Exemples d'imports de variables reprenant la structure de package donnée en page 17 :

```
from packageniv1.moduleniv1 import vp1m1
from packageniv1.packageniv2 import vp2
from packageniv1.packageniv2.moduleniv2 import vp2m2
from packageniv1 import vp1
```

Imports relatifs dans un package

Les imports entre les modules qui constituent un même package peuvent utiliser une notation relative, en commençant l'expression d'import par un point qui indique le niveau de package

19 Des restrictions sur les noms globaux importés par le * peuvent être mises en place, soit en faisant commencer les noms qui ne doivent pas être importés par un tiret souligné, soit - et c''est la méthode la plus propre - en définissant au début du module une variable globale `__all__` qui contient dans une liste de chaînes de caractères les noms qui sont à importer par la notation *.

du module courant, chaque point suivant faisant remonter d'un niveau dans la hiérarchie des packages. Cela permet, dans un package volumineux, de rendre les modules proches indépendants d'une restructuration de la hiérarchie du package.

Dans notre exemple de la page 17, le module **moduleniv2_2** s'est vu ajouter une instruction d'import relatif pour accéder à la variable **vp2m2** du module **moduleniv2**.

On a la structure (simplifiée) de répertoires et fichiers suivante :

```
├── packageniv1                                    variables définies
│   ├── packageniv2
│   │   └── moduleniv2.py                          vp2m2=8
│   ├── packageniv2_2
│   │   ├── moduleniv2_2.py                        vp22m2=12
```

L'import a été ajouté, ainsi qu'un affichage de la variable importée :

```
print("chargement packageniv1.packageniv2_2.moduleniv2_2")
vp22m2 = 12
from ..packageniv2.moduleniv2 import vp2m2
print(vp2m2)
```

Et à l'exécution, on retrouve le chargement des modules

```
>>> import packageniv1.packageniv2_2.moduleniv2_2
chargement packageniv1.__init__
chargement packageniv1.packageniv2_2.__init__
chargement packageniv1.packageniv2_2.moduleniv2_2
chargement packageniv1.packageniv2.__init__
chargement packageniv1.packageniv2.moduleniv2
8
```

> **Inconvénient** : un module utilisant l'import relatif n'est plus utilisable comme module principal.

Imports croisés

Il peut arriver que deux modules **A** et **B** doivent s'importer l'un-l'autre, ayant besoin d'accéder aux noms qu'ils définissent chacun de leur côté.

Si les deux modules **A** et **B** ont des éléments communs, il est possible de les placer dans un troisième module **C**, importé par les deux autres.

Si le module **A** n'a pas besoin des définitions de **B** immédiatement pour sa propre définition, il peut alors utiliser l'import de **B** en tant que module et passer par **B.nom** pour accéder au contenu. Si besoin, l'import de **B** peut être déplacée à la fin du module **A**, afin d'être sûr que **A** réalise complètement sa définition avant de demander les définitions de **B**.

5.b - Résolution des noms

La *résolution des noms vers les objets est faite à l'exécution*, en utilisant une notation pointée, par exemple **scene.vehicule.vitesse**. Les noms intermédiaires avant chaque point correspondent à des objets qui fournissent chacun un espace de noms dans lequel est recherché le nom suivant, jusqu'au nom final qui est associé à l'objet que l'on manipule.

Règle LGB

Le **premier nom** est recherché dans les *espaces de noms implicites* — il faut bien commencer quelque part — en parcourant dans l'ordre les espaces des noms *locaux* (à la fonction ou méthode), puis *globaux* (au module) et enfin *builtins* (standards). D'où le nom **LGB** pour l'ordre de recherche *Local Global Builtin* (on trouve aussi en français LGI pour Local Global Interne).

Dans notre exemple, **scene** est recherché en suivant la règle LGB.

Résolution dans les espaces de noms

Les **noms pointés** (ceux en partie droite d'un point) sont des noms à rechercher dans l'espace de noms de l'objet résolu avec ce qui est avant le point.

Si un nom correspond à un *module*, l'espace de noms de recherche est simplement l'*espace des noms globaux* de ce module. Si un nom correspond à un *objet*, on recherche d'abord dans l'*espace de noms de l'objet* lui-même, puis dans les *espaces de noms des classes* dont il hérite en remontant l'arbre d'héritage.

Dans notre exemple, `vehicule` est recherché dans l'espace de noms correspondant à `scene`, puis `vitesse` est recherché dans l'espace de noms de `vehicule`.

Nous verrons plus loin qu'il est possible de redéfinir la méthode de résolution de noms sur un objet et de définir des méthodes d'accès automatiquement appelées pour certains noms (voir *Contrôle d'accès aux attributs*, page 52).

6 - Affectation

En Python l'affectation consiste à **associer un nom** dans un espace de noms **avec un objet**. Il n'y a *pas de duplication* de l'objet affecté, juste la création d'une *nouvelle référence* (voir *Références sur objets,* page 27).

6.a - Règles d'affectation

Lorsqu'on accède à une variable en lecture, la résolution de noms présentée précédemment s'applique (règle LGB, espaces de noms). Par contre, lorsqu'on veut affecter une variable il faut connaître les règles suivantes :

- Une affectation effectuée au **niveau général d'un module** crée la variable dans l'*espace de noms des globales* du module.
- Une affectation effectuée **dans le corps d'une fonction/méthode** crée la variable dans l'*espace de noms des locales* de celle-ci.
    - Pour pouvoir, à l'intérieur d'une fonction, modifier par affectation (ou créer) une variable globale au module de cette fonction, il faut utiliser la directive `global nomvariable` qui permet de le préciser au compilateur d'aller directement manipuler la variable dans cet espace de noms.
    - Si une fonction est définie dans le corps d'une autre fonction, et si elle doit affecter une variable locale de cette autre fonction, il faut utiliser la directive `nonlocal nomvariable` qui permet de préciser au compilateur d'aller manipuler la variable dans l'espace de nom des locales de la fonction englobante. Si la fonction interne ainsi définie continue d'exister après l'appel (qu'elle soit retournée ou bien affectée à une variable globale), un système de fermeture (closure) est mis en place pour capturer le contexte de la fonction englobante, contexte qui peut ensuite continuer à être modifié par l'instruction d'affectation.
- Une affectation **spécifiant un attribut d'un objet** crée la variable (le nom) dans l'espace de noms de cet objet (les noms précédant les points de l'expression pointée sont résolus, et le dernier nom est créé dans l'espace de noms juste avant). En reprenant l'exemple précédent, pour l'instruction d'affectation `scene.vehicule.vitesse=45`, on a la résolution de `scene.vehicule` vers un objet, et `vitesse` est créée dans l'espace de noms de cet objet.

Pour assurer tout de même un bon niveau de performances, certains objets internes ne permettent pas de créer dynamiquement de nouveaux noms dans leur espace de noms ni de supprimer ceux existants.

Un attribut affecté dans l'espace de nom d'un objet est spécifique à cet objet. Pour qu'il soit visible par tous les objets d'une classe, il faut réaliser l'affectation dans l'espace de noms de la classe. Par ailleurs, les règles de résolution des attributs font qu'un nom défini dans l'espace de noms d'un objet masque un nom similaire défini dans l'espace de noms de la classe de cet objet.

Illustrations

Dans l'exemple ci-dessous une variable, une fonction et une classe sont créées au niveau du module **globmodules**, et apparaissent dans son espace de noms.

```
# file: globmodules.py
a = 4
def f(x): return x+1
class C: pass
print(dir())
```

```
$ python3 globmodules.py
['C', '__builtins__', '__cached__', '__doc__', '__file__', '__name__',
'__package__', 'a', 'f']
```

L'exemple suivant illustre la réaffectation d'une variable globale, modifiée à l'intérieur d'une fonction, en ayant spécifié la directive **global**, ainsi que le masquage d'une variable globale par une variable locale de même nom.

```
# file: globfonctions.py
modifiee = "original"
masquee = "original"
def f():
    global modifiee
    modifiee = "changee"
    masquee = "changee"
    print("Dans f(), modifiee:", modifiee, "masquee:", masquee)

print("Avant f(), modifiee:", modifiee, "masquee:", masquee)
f()
print("Avant f(), modifiee:", modifiee, "masquee:", masquee)
```

```
$ python3 globfonctions.py
Avant f(), modifiee: original masquee: original
Dans f(), modifiee: changee masquee: changee
Avant f(), modifiee: changee masquee: original
```

Une erreur courante concernant les variables globales est illustrée ci-dessous :

```
# file: globerrinc.py
cptr = 0
def incrementer():
    cptr = cptr + 1
incrementer()
```

```
$ python3 globerrinc.py
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "globerrinc.py", line 5, in <module>
    incrementer()
  File "globerrinc.py", line 4, in incrementer
    cptr = cptr + 1
UnboundLocalError: local variable 'cptr' referenced before assignment
```

L'association du message d'erreur avec la cause, si elle est logique, n'est pas immédiate à comprendre. La variable **cpt** étant la cible d'une instruction d'affectation, en absence de la directive **global** elle est considérée lors de la compilation comme un nom local de la fonction. Or la partie droite de l'affectation utilise cette variable, qui n'existe pas au niveau local. D'où l'erreur indiquant l'utilisation de la variable avant qu'elle n'existe localement par une affectation.

Enfin, le dernier exemple montre la capture du contexte (fermeture/« closure ») par une fonction et la possibilité de modifier ce contexte en utilisant la directive **`nonlocal`** pour une variable.

```
# file: nonlocalinc.py
def incrementeur(n):
    actuel = 0
    def compte():
        nonlocal actuel
        actuel = actuel + n
        return actuel
    return compte

inc3 = incrementeur(3)
print("inc3:", inc3())
print("inc3:", inc3())
inc5 = incrementeur(5)
print("inc5:", inc5())
print("inc5:", inc5())
print("inc3:", inc3())
```

```
$ python3 nonlocalinc.py
inc3: 3
inc3: 6
inc5: 5
inc5: 10
inc3: 9
```

6.b - Suppression

Comme une affectation est simplement la définition d'un nom dans un espace de noms et son association avec un objet, il existe une opération inverse, **`del nomvariable`**, qui permet la destruction du nom et de l'association qui va avec.

```
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__']
>>> one = 1
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'one']
>>> del one
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__']
```

Ceci fonctionne, quel que soit l'objet référencé et, quelle que soit la façon dont le nom a été défini (affectation, définition de fonction, définition de classe, importation).

6.c - Réaffectation

La réaffectation consiste simplement à associer un autre objet à un nom auquel est déjà affecté un objet. Le système de gestion de la mémoire se charge de mettre à jour les compteurs de références de l'ancien objet affecté et du nouveau, et de la suppression de l'ancien objet si nécessaire.

6.d - Typage des variables

Tel qu'elles sont définies les variables ne sont que des accès vers les objets, elles ne portent aucune autre information. **Le type d'une variable est le type de l'objet référencé**.

Ceci a une implication importante : **les variables sont dynamiquement typées**. La réaffectation permet en effet d'associer à un nom n'importe quel autre objet.

6.e - Affectations multiples

Il est possible d'affecter la même valeur à plusieurs variables en une instruction.

```
>>> a = b = c = 0
```

```
>>> a
```

```
0

>>> b
0
```

```
>>> c
0
```

Il est possible d'affecter différentes valeurs à différentes variables en une instruction. Cela passe par l'utilisation d'une séquence de valeurs[20] en partie droite de l'affectation, et l'affectation des éléments un à un aux noms en partie gauche.

```
>>> a,b,c = 4,"quatre",4.0
>>> a
4
>>> b
'quatre'
>>> c
4.0
```

```
>>> a,b,c = ["contenu","de","liste"]
>>> a
'contenu'
>>> b
'de'
>>> c
'liste'
```

Grâce à ce fonctionnement, l'échange de deux variables s'écrit simplement : `a,b = b,a`.

L'utilisation d'une étoile préfixant la dernière variable en partie gauche de l'affectation permet d'indiquer d'affecter "ce qui reste" de la séquence de valeurs en partie droite sous forme de séquence `list`. Ce mécanisme fonctionne avec toute valeur itérable en partie droite (voir *Boucle sur itérable*, page 32).

```
>>> tete,*queue = "x","y","z","t",5
>>> tete
'x'
>>> queue
['y', 'z', 't', 5]
>>> lst = [0,1,2,3,4,5,6,7,8,9]
>>> tete, *queue = lst
>>> tete
0
>>> queue
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> d = dict(a=1,b=2,c=3,d=4)
>>> tete,*queue = d
>>> tete
'a'
>>> queue
['c', 'b', 'd']
>>> tete,*queue = range(10)
>>> tete
0
>>> queue
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

6.f - Affectation augmentée

Cette notation permet de réaliser directement une opération avec l'élément en partie gauche de l'affectation, et de stocker le résultat dans cet élément. Elle permet d'utiliser certains opérateurs en les combinant avec l'opérateur d'affectation : `+=`, `-=`, `*=`, `/=`, `//=`, `%=`, `**=`, `>>=`, `<<=`, `&=`, `^=`, `|=`. L'*expression en partie gauche n'est évalué qu'une fois*, c'est donc plus efficace lorsque l'expression est longue (et aussi plus lisible), mais on s'en sert aussi souvent pour signifier un simple incrément.

```
>>> a = 1
>>> a = a + 1
>>> a
2
>>> a += 1
>>> a
3
```

```
>>> data=[0,0,[0,{'truc':8,'machin':12}]]
>>> data[2][1]['machin'] *= 3
>>> data
[0, 0, [0, {'machin': 36, 'truc': 8}]]
```

> **Attention** : si pour les types immutables l'opération d'affectation augmentée est équivalente aux deux opérations séparées, ça n'est pas obligatoirement le cas pour les types de données mutables. L'affectation augmentée est en effet une opération en tant que telle, qui peut être redéfinie (voir *Redéfinition de l'affectation augmentée*, page 51), et certains types de données mutables réalisent des transformations directes de l'objet original sans passer par une réaffectation.

```
>>> lst = [1,2,3]
```

```
>>> lst2 = lst
```

20 Lorsqu'on utilise une notation simplement avec des virgules, Python construit une séquence de type `tuple`.

```
>>> lst = lst + [4]
>>> lst
[1, 2, 3, 4]
>>> lst2
[1, 2, 3]
>>> lst = [1,2,3]
```

```
>>> lst2 = lst
>>> lst += [4]
>>> lst
[1, 2, 3, 4]
>>> lst2
[1, 2, 3, 4]
```

7 - Références sur objets

Le fonctionnement basé sur l'utilisation de références sur des objets se retrouve partout en Python. Comme nous l'avons vu, une variable est un nom qui référence un objet correspondant à une valeur d'un type donné. Un identificateur de fonction est un nom qui référence un objet fonction, celle-ci comporte divers attributs dont un qui référence le bytecode nécessaire à son exécution. Une classe est un nom qui référence un objet classe. Un module importé est un nom qui référence un objet module. Etc.

Les conteneurs Python de base ne stockent pas de valeurs, ils stockent des références vers les objets qui correspondent à ces valeurs. Les valeurs passées en paramètres aux fonctions et méthodes sont simplement des duplications des références vers les objets correspondants.

Un programme Python en cours d'exécution est finalement un grand graphe en cours d'évolution.

Pour savoir si deux références correspondent au même objet, on peut utiliser soit l'opérateur `is` (ou `is not`) qui réalise un **test d'identité** sur les objets, soit la fonction standard `id()` qui retourne un *identificateur* de l'objet (en fait son adresse mémoire en cpython).

7.a - Objets immutables

Tant qu'il est *impossible de modifier la valeur contenue* dans l'objet qui est référencé, en Python on dit que l'objet est **immutable**, le fonctionnement est similaire à ce que l'on retrouve dans la plupart des langages, le fait qu'il faille passer par une référence est transparent et n'a aucune implication pour le programmeur.

Une fois qu'une expression a été évaluée et a produit un objet immutable correspondant à la valeur, on est sûr que celui-ci ne changera pas. Si on a récupéré une référence sur un objet immutable, on est sûr qu'il restera toujours le même. Si une expression de transformation est appliquée sur un objet immutable, le résultat ne pourra être qu'un nouvel objet distinct.

La plupart des valeurs de base du langage sont immutables, les valeurs numériques (`int`, `float`, `complex`), les booléens (`bool`), la valeur « nulle » ou indéfini (constante `None`), les chaînes de caractères (`str`), les séquences d'octets (`bytes`). Il en est de même pour les conteneurs séquences simples indexées (`tuple`[21]) et les ensembles figés (`frozenset`).

Illustration

Du fait que l'affectation ne fait que créer une *nouvelle référence* et qu'une nouvelle expression[22] peut créer un *nouvel objet* de valeur égale, pour modifier une variable référençant une valeur immutable il faut passer par une réaffectation de cette variable à une nouvelle valeur, car la valeur elle-même ne peut pas être changée.

```
>>> a = 856
>>> b = a
```

```
>>> a is b
True
```

21 Le type tuple est beaucoup utilisé en interne par Python.

22 On n'a spécifiquement *pas* utilisé d'entiers de faible valeur, pour lesquels cpython génère des singletons qui sont réutilisés partout où ces valeurs sont nécessaires (`1000 is 2*500` est faux, car on a bien deux objets entiers créés avec la même valeur, mais en raison de cette optimisation interne du langage `4 is 2*2` est vrai, car l'objet entier pour la valeur 4 est automatiquement réutilisé).

```
>>> c = 856
>>> a is c
False
>>> a == c
True
```

```
>>> id(a),id(b),id(c)
(14216976, 14216976, 14635216)
>>> a = 1000
>>> a,b,c
(1000, 856, 856)
```

Cas un peu spécial des constantes du langage, `None`, `True` et `False`, qui sont des singletons automatiquement utilisés lors du calcul de certaines expressions.

```
>>> id(None)
8761216
>>> id(True)
9136256
>>> id(False)
9136288
>>> vrai = 1 == 1
```

```
>>> vrai is True
True
>>> id(vrai)
9136256
>>> id(vrai) == id(True)
True
```

7.b - Objets mutables

Lors qu'il est possible de *modifier la valeur contenue dans un objet*, en Python on dit que l'objet est **mutable**, cela doit être absolument pris en compte lorsqu'on le manipule. En effet, toute nouvelle référence créée dans le programme, par exemple une simple affectation ou un passage de paramètre, est un accès possible pour modifier la valeur dans l'objet — **il n'y a pas de « const » en Python**.

Cette prise en compte passe par *une définition claire et une connaissance des effets de bord* que peuvent faire certaines fonctions ou méthodes, et si nécessaire une *duplication des données* lorsque l'on a besoin de les transformer en voulant éviter d'altérer la valeur d'origine.

De nombreux types conteneurs standards sont mutables, les listes[23] (`list`), les dictionnaires qui gèrent des associations clés/valeurs (`dict`), les ensembles (`set`), les tableaux dynamiques d'octets (`bytearray`). Le module `copy` fournit des fonctions `copy()` et `deepcopy()` pour réaliser des copies d'objets et de conteneurs, de premier niveau ou bien en profondeur. Nous verrons que certaines expressions sur les conteneurs génèrent aussi des copies de premier niveau.

La plupart des classes qui fournissent des services évolués sont aussi mutables, entre autres les classes définies simplement par les programmeurs.

Illustration

Que l'affectation avec des valeurs de types mutables se limite, comme avec les types immutables, à la création d'une nouvelle référence pour le nouveau nom. Mais que les modifications apportées sont visibles via toutes les variables qui référencent l'objet correspondant. Exemple avec une valeur de type `list`.

```
>>> lst = [1,-3,8]
>>> lst2 = lst
>>> lst2 is lst
True
>>> lst[1] = 1000
```

```
>>> lst
[1, 1000, 8]
>>> lst2
[1, 1000, 8]
```

8 - Contrôle de flux

Les instructions de contrôle de flux de base se limitent aux instructions de test `if`, boucles `while` et `for`, et de rupture `break` et `continue`. Et c'est tout. La gestion des itérateurs par Python permet de prendre en compte beaucoup de cas de boucles d'une façon standardisée et propre.

On dispose bien sûr de la *définition de fonctions*, qui permet d'avoir des sections de code réutilisables à différents endroits dans les programmes.

23 Les « listes » Python sont en fait des tableaux dynamiques.

On trouve enfin les instructions de contrôles de flux liées aux exceptions avec **`try`**, **`except`**, **`finally`** et **`raise`**, ainsi que le cas particulier du bloc de contexte géré avec **`with`** qui automatise des opérations de sortie de bloc d'instructions, que ce soit de façon normale ou sur exception.

8.a - Blocs d'instructions

La définition des blocs d'instructions en Python se fait via l'**indentation**. Il n'y a *pas de mot clé ni de caractère spécial* pour indiquer le début ou la fin du bloc. C'est l'indentation d'une ligne, puis des lignes suivantes au même niveau, jusqu'à une ligne qui soit indentée en retrait, qui *définissent visuellement et syntaxiquement le début et la fin du bloc.*

> **Attention** : une erreur courante est liée au mélange d'espaces et de tabulations dans le code, ce qui donne une indentation *visuellement identique*, mais *syntaxiquement différente* en raison du nombre différent de caractères d'indentation.
>
> Les **bons usages** en Python[24] bannissent l'utilisation du caractère tabulation et préconisent **4 espaces par niveau d'indentation** ; tout éditeur correct devant permettre un tel réglage.

Un exemple de code correctement indenté, qui présente différentes structures de contrôle de flux avec les blocs d'instructions correspondants :

```
def affiche_collisions():
    """Recherche des uids doublons dans /etc/passwd

    Affiche les uid doublons trouvés et les login correspondant.
    Retourne un indicateur booléen que des collisions ont été trouvées.
    """
    uidvus = {}
    collisions = 0
    with open("/etc/passwd") as f:
        for line in f:
            items = line.split(":")
            login = items[0]
            uid = int(items[2])
            if uid in uidvus:    # uid déjà connu, on indique le doublon.
                print(login, "! Déjà vu uid", uid, "pour", uidvus[uid])
                collisions += 1
            else:                # uid inconnu, on le référence.
                uidvus[uid] = login
    print("Trouvé", collisions, "collisions")
    return collisions != 0
```

Remarque : l'indentation propre est ce que l'on fait naturellement dans les autres langages pour avoir un code plus lisible, là c'est aussi une obligation syntaxique. Pour les inconditionnels, il est possible de préciser plus visuellement ces blocs en utilisant des commentaires qui reprennent les syntaxes courantes, par exemple avec `#{` et `#}` ou avec `#begin` et `#end`. On peut même traduire en français `#début` et `#fin`.

L'inconvénient majeur avec l'utilisation de l'indentation pour délimiter les blocs est lié à la transmission de code via des outils qui considèrent toute séquence de caractères blancs comme un seul séparateur. Typiquement le web avec le HTML. La solution, valable pour tous les langages d'ailleurs, est d'encadrer le code source par des balises `<pre>` `</pre>` afin d'indiquer au moteur de rendu HTML de conserver la mise en forme choisie, avec des blancs significatifs. Pour la transmission par courrier électronique, on préfère l'envoi de pièces jointes, qui garantit que tout le contenu du fichier est bien préservé.

24 Voir le PEP8 « Style Guide for Python Code » : http://www.python.org/dev/peps/pep-0008/

Instruction vide

L'instruction `pass` est une instruction qui ne fait rien. Si un bloc d'instructions doit être vide, comme il doit y avoir une instruction indentée pour que le bloc existe, on peut y placer une unique instruction `pass`. C'est souvent utilisé pour définir une classe ou une fonction vide.

```
# Une jolie boucle sans fin.
while True:
    pass
class Essai: pass
```

Instructions composées

Les instructions composées, qui sont abordées ci-dessous, relient *une instruction de contrôle de flux* à *une instruction ou un bloc d'instructions*. L'instruction de contrôle de flux se termine par un caractère deux-points (`:`), qui est suivi soit d'une unique instruction sur la même ligne, soit d'un bloc d'instructions indenté.

La forme avec bloc d'instructions indenté est à préférer, même lorsqu'il n'y a qu'une seule instruction dans le bloc. Ceci pour d'évidentes raisons de lisibilité, mais aussi pour faciliter l'identification des erreur détectées en séparant bien la ligne d'introduction de l'instruction composée du bloc d'instructions qui est lié.

Instruction sur la même ligne (lorsqu'il n'y a qu'une instruction) :

```
a = 10
while a<100: a = a + 1
print(a)
if a==100: a = a+1
else: a = a-1
```

Instruction(s) dans un bloc indenté (à préférer, même avec une seule ligne !) :

```
a = 10
while a<100:
    a = a + 1
print(a)
if a==100:
    a = a+1
else:
    a = a-1
```

8.b - Test

L'instruction de test `if condition:` permet d'introduire un bloc d'instructions dont l'exécution est liée à l'évaluation de la condition logique. Elle peut éventuellement se poursuivre avec une série de `elif condition:`, introduisant des conditions alternatives avec les blocs d'instructions à exécuter. Et elle peut se terminer avec un `else:` avec le bloc d'instructions pour tous les autres cas.

Exemple d'utilisation :

```
# Racines réelles d'une équation du second degré a×x²+b×x+c=0
delta = b ** 2 - 4 * a * c
if delta == 0:
    x = -b / (2 * a)
    resultat = x
elif delta > 0:
    x1 = (-b - sqrt(delta)) / (2 * a)
    x2 = (-b + sqrt(delta)) / (2 * a)
    resultat = (x1, x2)
else: # delta < 0
    resultat = None
```

En cas de tests `if`/`elif` consécutifs, les tests s'arrêtent à la première condition vérifiée, avec l'exécution du bloc d'instructions correspondant.

8.c - Expression conditionnelle

Équivalent de l'opérateur ternaire *(condition?x:y)* que l'on trouve dans d'autres langages, en Python on utilise le **if** directement dans une expression sous la forme **x if condition else y**. L'évaluation de l'expression conduit à l'évaluation de *x* si *condition* est vraie, et à l'évaluation de *y* sinon. Malgré l'ordre d'écriture, *condition* est bien évaluée en premier, et son résultat ne provoque l'évaluation que d'une seule des deux expressions.

Écriture d'une expression de la valeur absolue[25] :

```
>>> x = -7
>>> y = -x if x < 0 else x
>>> y
7
```

Si x et y doivent être simplement des constantes **True**/**False**, l'expression s'écrit plus simplement en tant qu'expression logique booléenne (voir *Logique*, page 47) sans avoir à utiliser une expression conditionnelle. Par exemple pour avoir un indicateur booléen qu'une valeur est bien entre 10 et 20 :

```
>>> x = 5
>>> z = True if 10 <= x < 20 else False  # expression conditionnelle
>>> z
False
>>> z = 10 <= x < 20                     # expression booléenne simple
>>> z
False
```

Ancienne façon avec l'indexation

On trouve encore une ancienne façon de réaliser l'opérateur ternaire, consistant à utiliser une séquence **tuple**[26] de deux valeurs que l'on indexe suivant une valeur booléenne :

```
(y,x)[bool(condition)]
```

L'utilisation du transtypage des booléens **False** et **True** en index entiers avec les valeurs 0 et 1 fait que la valeur si vrai doit être en seconde position.

Exemple :

```
>>> ("si faux","si vrai")[True]
'si vrai'
>>> ("si faux","si vrai")[False]
'si faux'
```

```
>>> ("si faux","si vrai")[1]
'si vrai'
>>> ("si faux","si vrai")[0]
'si faux'
```

Ceci n'est toutefois pas exactement équivalent à l'expression conditionnelle **if**, car la construction du tuple provoque l'*évaluation des deux expressions*.

8.d - Boucle conditionnelle

L'instruction de boucle *tant que*, **while condition:**, permet de répéter l'exécution du bloc d'instructions tant que la condition logique est évaluée à vrai.

```
>>> x = 5
>>> while x > 0:
...     print("x:", x)
...     x = x - 1
...
x: 5
```

```
x: 4
x: 3
x: 2
x: 1
>>> x
0
```

25 C'est pour l'exemple, il existe une fonction **abs()** dans l''espace de noms standard builtins.

26 Ou tout autre conteneur indexable avec 0/1 ou False/True (liste, dictionnaire, etc). En cas de valeurs fixes, l'utilisation d''un tuple a l'avantage qu''il est construit une seule fois comme donnée statique.

8.e - Boucle sur itérable

L'instruction de boucle *pour*, `for variable in iterable:`, permet de parcourir les valeurs contenues ou générées dynamiquement par l'objet itérable, en affectant tour à tour chacune de ces valeurs à la variable :

```
>>> for v in ["un", "bon", "mot"]:
...     print(v)
...     print("-" * 10)
...
un
----------
bon
----------
mot
----------
>>> v
'mot'
```

La variable de boucle continue d'exister après la boucle, en conservant la dernière valeur obtenue lors de l'itération. S'il n'y a pas eu d'itération, la variable contient ce qu'elle contenait avant la boucle (ou n'existe pas si elle n'a jamais été affectée avant).

On peut aussi utiliser plusieurs variables dans la boucle, pour autant que les valeurs produites par l'itérateur contiennent plusieurs éléments :

```
>>> lst = [(1,1), (4.2,5), (8,9.1)]
>>> for x,y in lst:
...     print("Coordonnées:",x, y)
...     print("Distance:", hypot(x,y))
...
Coordonnées: 1 1
Distance: 1.4142135623730951
Coordonnées: 4.2 5
Distance: 6.529931086925803
Coordonnées: 8 9.1
Distance: 12.116517651536682
```

La boucle sur itérable effectuant une affectation sur les variables de boucle, il est possible d'y utiliser certaines syntaxes spéciales :

```
>>> for a, *b in ([1,2,3,4], [1,2], [8,7,6,5,4], [4]):
....     print(a, b)
....
1 [2, 3, 4]
1 [2]
8 [7, 6, 5, 4]
4 []
```

Généralisation des itérables

Cette catégorie de boucle repose sur l'utilisation d'une **interface itérable standard**, permettant à tout objet de définir qu'il est itérable et d'implémenter la façon dont les différentes valeurs sont produites au fur et à mesure des itérations (voir *Redéfinition de l'itération*, page 51).

Une catégorie particulière d'objets itérables sont les **générateurs**, qui construisent dynamiquement les valeurs d'itération au fur et à mesure des besoins.

En mode interactif, les générateurs étant des objets de haut niveau, si on veut voir les valeurs qu'ils génèrent il faut les transtyper vers un type conteneur, comme une liste qui consomme les valeurs produites pour se construire.

```
>>> range(10)
range(0, 10)
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> enumerate("Python")
<enumerate object at 0x2224190>
>>> list(enumerate("Python"))
[(0, 'P'), (1, 'y'), (2, 't'), (3, 'h'), (4, 'o'), (5, 'n')]
```

Générateur de suites entières

La fonction builtin standard `range()` retourne un générateur produisant une suites de nombres entiers, en pouvant choisir début, fin et pas. Par défaut le début commence à zéro et le pas est de un. La valeur de fin n'est pas comprise dans la suite. On peut aussi utiliser des valeurs négatives.

```
>>> list(range(7))
[0, 1, 2, 3, 4, 5, 6]
>>> list(range(3, 7))
[3, 4, 5, 6]
>>> list(range(3, 7, 2))
[3, 5]
```

```
>>> list(range(-10, -5, 1))
[-10, -9, -8, -7, -6]
>>> list(range(-5, -10, -1))
[-5, -6, -7, -8, -9]
>>> list(range(8, 2))
[]
```

Générateur d'énumération index, valeur

La fonction builtin standard `enumerate()` utilise un itérable en paramètre, ainsi qu'une valeur d'index initial (par défaut 0). Elle retourne un générateur produisant une suite de couples (tuples) contenant en première position un index incrémenté à partir de zéro et en deuxième position les valeurs produites par l'itérable donné.

```
>>> lst = ["un", "deux", "mille", "beaucoup"]
>>> for i,v in enumerate(lst):
...     print(i, v)
...
0 un
1 deux
2 mille
3 beaucoup
```

Ce générateur est couramment utilisé pour connaître l'index de la position de l'élément sur lequel on travaille, afin de pouvoir faire un remplacement dans le conteneur avec une valeur modifiée.

Expression générateur

Il est possible d'exprimer les valeurs à générer sous la forme d'une expression en compréhension (voir *Générateurs en compréhension*, page 50 pour plus de détails), qui produisent les valeurs calculées au fur et à mesure des besoins.

```
>>> for v in (x ** 4 for x in (1, 5, 9, 12)):
...     print(v, end=" ")
...
1 625 6561 20736
```

Pratique lorsque les valeurs à générer sont de simples calculs qui peuvent s'exprimer avec une ou plusieurs boucles et éventuellement un test de filtrage.

```
>>> for v in (y/x for x in (-2, -1, 0, 1, 2) for y in (10, 20, 30) if x!=0):
...     print(v, end=" ")
...
-5.0 -10.0 -15.0 -10.0 -20.0 -30.0 10.0 20.0 30.0 5.0 10.0 15.0
#[     -2       ] [       -1        ] [      1      ] [     2      ]  boucle sur x
#[10   20    30    10   20    30    10   20   30   10  20   30]  boucle sur y
#la valeur x==0 a été filtrée pour éviter la division par zéro
```

Fonctions générateur

Une fonction générateur[27] est une fonction dont l'exécution peut être suspendue lorsqu'elle fournit une valeur, puis reprise pour fournir une valeur suivante, etc jusqu'à ce qu'elle se termine. L'instruction `yield` permet d'indiquer la valeur fournie à un moment donné et de suspendre l'exécution. On utilise aussi le terme de « *coroutines* » ; ce mécanisme peut être utilisé pour simuler un fonctionnement multitâche coopératif et est à la base de librairies simulant les tubes (pipes) entre des fonctions de traitement sur des flux de données[28].

```
>>> def gene5():
...     yield 1
...     yield 2
...     yield 3
...     yield 4
...     yield 5
```

```
...
>>> g5 = gene5()
>>> for v in g5:
...     print(v, end=" ")
...
1 2 3 4 5
```

27 Voir le PEP342 « Coroutines via Enhanced Generators » : http://www.python.org/dev/peps/pep-0342/

28 Voir les librairies tiers calabash, chut, pyxshell, plumbum, etc.

L'appel à la fonction provoque la *création d'un générateur* basé sur cette fonction, qui capture le contexte de la fonction au moment où elle est appelée, et qui est u*tilisable partout où un itérable est accepté.*

On aura généralement plutôt des boucles dans les fonctions générateurs :

```
>>> def carres(n):
...     i = 1
...     while i < n:
...         yield i ** 2
...         i = i + 1
...
>>> for v in carres(10):
...     print(v, end=" ")
...
1 4 9 16 25 36 49 64 81
```

Il est possible de contrôler le générateur ainsi créé en utilisant sa méthode `.send()` qui permet de lui *transmettre une valeur* qui est récupérée dans le générateur comme résultat de l'exécution du `yield` et de renvoyer ensuite une *valeur de retour* de cette transmission par la valeur du `yield` suivant.

Voici un exemple relativement simple de générateur pilotable qui produit à chaque itération un tuple contenant un entier (incrémenté à chaque itération) et une chaîne (a priori toujours la même). Il est possible de modifier la chaîne retournée par les itérations suivantes en envoyant une nouvelle chaîne via `.send("chaine")`, et aussi de demander l'arrêt propre du générateur en envoyant une chaîne spéciale `.send("arret")`.

```
# file: generateurpilote1.py
def pilote(chaine):
    val = 0
    fini = False
    res = yield (val, chaine)
    while not fini:
        if res is None:         # "normal", production prochaine valeur
            val = val + 1
            res = yield (val, chaine)
        elif res == "arret":    # un appel gene.send("arret") a été fait
            fini = True
            res = yield "ok arret"    # valeur retour du send()
        else:                   # un appel gene.send("qqchose") a été fait
            chaine = res
            res = yield "ok changé"   # valeur retour du send()

print("début")
gene = pilote("bosse")
for i,v in gene:
    print(i, v)
    if i == 3:
        r = gene.send("trou")      # change la chaîne générée
        print("reponse:", r)
    if i == 7:
        r = gene.send("arret")     # demande l'arrêt du générateur
        print("reponse:", r)
print("fin")
```

```
$ python3 generateurpilote1.py
début
0 bosse
1 bosse
2 bosse
3 bosse
reponse: ok changé
4 trou
5 trou
6 trou
7 trou
reponse: ok arret
fin
```

Une solution pour structurer le code de la fonction générateur se base sur la gestion d'une file dans laquelle sont ajoutées les réponses à fournir aux appels à `.send()`.

```
# file: generateurpilote2.py
def pilote(chaine):
    val = 0
    fini = False
    reponses = []
    while not fini or reponses:
        if reponses:
            res = yield (reponses.pop(0))
        else:
            res = yield (val, chaine)
        if res is None:         # "normal", production prochaine valeur
            val = val + 1
        elif res == "arret":    # un appel gene.send("arret") a été fait
            fini = True
            reponses.append("ok arret")     # valeur retour du send()
        else:                   # un appel gene.send("qqchose") a été fait
            chaine = res
            reponses.append("ok changé")    # valeur retour du send()

# L'appel au générateur ne change pas.
```

Il est aussi possible de transmettre une exception au générateur via sa méthode `.throw()`, exception qui est levée dans le cadre du générateur ; ainsi que de forcer son arrêt via sa méthode `.close()`.

Itération sur fichiers textes

Les objets fichiers textes (voir *Fichiers texte*, page 53) supportent l'interface d'itération, qu'ils implémentent en produisant à chaque itération la ligne suivante lue dans le fichier. Il est extrêmement courant d'avoir l'écriture suivante :

```
f = open("index.html")
for line in f:
    # traitement
    pass
f.close()
```

Outils itérateurs

La fonction standard builtin `iter()` permet de construire un itérateur à partir d'un objet. Avec un paramètre, elle permet de parcourir un objet séquence qui n'implémenterais pas le protocole d'itération en récupérant les valeurs par leur index (voir *Redéfinition des séquences*, page 51). Avec deux paramètres, elle permet d'itérer en faisant un appel sur le premier paramètre, jusqu'à ce que la valeur d'itération retournée par cet appel soit égale à la valeur sentinelle donnée en deuxième paramètre — l'exemple typique d'utilisation, donné dans la documentation Python, est l'appel à la méthode de lecture dans un fichier jusqu'à obtenir une valeur spécifique.

La fonction standard builtin `zip()` permet de construire un itérateur qui combine les éléments de même index issus de plusieurs séquences. L'itération s'arrête à la fin de la séquence la plus courte (voir `itertools.zip_longest()` pour aller jusqu'à la fin de la séquence la plus longue).

```
>>> for a in zip([1,2,3,4,5],['a','b','c','d','e'],
                 ['la','fonction','zip','en','action']):
...     print(a)
...
(1, 'a', 'la')
(2, 'b', 'fonction')
(3, 'c', 'zip')
(4, 'd', 'en')
(5, 'e', 'action')
```

Combinée avec l'opérateur * appliqué au passage de paramètres, zip() permet de remettre en place des listes zippées.

```
>>> l1,l2,l3 = zip(*[(1,'a','la'),(2,'b','fonction'),(3,'c','zip'),
                     (4,'d','en'),(5,'e','action')])
>>> l1
(1, 2, 3, 4, 5)
>>> l2
('a', 'b', 'c', 'd', 'e')
>>> l3
('la', 'fonction', 'zip', 'en', 'action')
```

Le module standard itertools fournit une collections d'itérateurs inspirés d'autres langages et adaptés en Python. Ils ne seront pas décrits ici, mais leurs noms peuvent déjà donner une idée de leurs fonctionnalités : count(), cycle(), repeat(), chain(), compress(), dropwhile(), groupby(), ifilter(), ifilterfalse(), islice(), imap(), starmap(), tee(), takewhile(), izip(), izip_longuest(), product(), permutations(), combinations(), combinations_with_replacement().

Un exemple d'utilisation :

```
>>> from itertools import permutations
>>> list(permutations([1,2,3]))
[(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]
```

8.f - Rupture de séquence

Les boucles while et for disposent de deux outils pour permettre de contrôler les itérations et d'une clause else: particulière introduisant un bloc d'instructions exécuté après la boucle.

Passage à l'itération suivante

L'instruction continue fait passer immédiatement à l'itération suivante de la boucle. Dans le cas d'une boucle conditionnelle, l'exécution repart sur l'évaluation de la condition de boucle. Dans le cas d'une boucle sur itérateur, l'exécution repart sur l'accès à l'élément de l'itération suivante.

```
for v in collection:
    if pas_interessant(v):
        continue  # passage au suivant
    # ici le traitement des cas intéressants
```

Arrêt en cours de boucle

L'instruction break fait sortir immédiatement de la boucle, sans passer par la clause else: s'il y en a une. L'exécution se poursuit après l'instruction composée de boucle.

```
for v in collection:
    if est_trouve(v):
        traitement(v)
        break
```

La levée d'une *exception*, si elle n'est pas stoppée à l'intérieur du corps de la boucle, provoque aussi un arrêt de cette boucle.

Bloc de sortie de boucle

La clause else: avec un bloc d'instructions peut être ajoutée aux l'instruction while et for. Elle n'est exécutée que lorsque l'on sort « normalement » de la boucle, c'est à dire par une évaluation de la condition logique à faux pour une boucle conditionnelle, et par l'arrivée à la fin des valeurs pour une boucle sur itérateur. Une sortie de boucle par une instruction de rupture de séquence break ne fait pas passer par le bloc else:.

Exemple avec une boucle while :

```
>>> lst = [1, 2, 3, 4, 5, 6, 7, 8]
>>> idx = 0
>>> while idx < len(lst):
...     if lst[idx] == 5:
...         index = idx
...         break
...     idx = idx + 1
... else:
...     index = None
...
>>> print(index)
4
```

```
>>> lst = [1, 2, 3, 4, 6, 7, 8]
>>> idx = 0
>>> while idx < len(lst):
...     if lst[idx] == 5:
...         index = idx
...         break
...     idx = idx + 1
... else:
...     index = None
...
>>> print(index)
None
```

S'il n'y a aucune itération (condition fausse dès avant la boucle), la clause **else** est exécutée (c'est une sortie normale de la boucle).

Exemple avec une boucle for :

```
>>> lst = [1, 2, 3, 4, 5, 6, 7, 8]
>>> for v in lst:
...     if v == 5:
...         trouve5 = True
...         break
... else:
...     trouve5 = False
...
>>> trouve5
True
```

```
>>> lst = [1, 2, 3, 4, 6, 7, 8]
>>> for v in lst:
...     if v == 5:
...         trouve5 = True
...         break
... else:
...     trouve5 = False
...
>>> trouve5
False
```

S'il n'y a aucune itération (aucune valeur disponible dans l'itérable), la clause **else** est exécutée (c'est une sortie normale de la boucle).

8.g - Faire un switch

Il n'y a pas d'instructions de sélection conditionnelle comme les switch/case que l'on rencontre dans d'autres langages. Il y a deux façons courantes de réaliser une telle sélection en Python.

Imbriquer des if/elif

Cette solution permet de placer une série de conditions, qui peuvent être plus que de simples tests d'égalité, avec les blocs d'instruction correspondants.

```
# Pour un montant, donne le taux et les bornes de la tranche d'imposition.
if 0 <= montant <= 5963:
    tranche = (0.0, 0, 5963)
elif  5963 < montant <= 11896:
    tranche = (5.5, 5963, 11896)
elif  11896 < montant <= 26420:
    tranche = (14.0,  11896, 26420)
elif  26420 < montant <= 70830:
    tranche = (30.0,  26420, 70830)
elif  70830 < montant <= 150000:
    tranche = (41.0,  70830, 150000)
elif  150000 < montant:
    tranche = (45.0,  150000, None)
```

Les conditions peuvent par exemple utiliser des reconnaissances d'expressions régulières en utilisant le module standard **re**.

```
# Identifier des balises HTML.
import re
if re.match("<h[0-6]>", chaine):
    categorie = "debut_heading"
elif re.match("/<h[0-6]>", chaine):
    categorie = "fin_heading"
elif chaine == "<b>":
    categorie = "debut_bold"
```

```
elif chaine == "</b>":
    categorie = "fin_bold"
```

Utiliser un dictionnaire

Cette seconde solution est privilégiée lorsque les blocs d'instructions liés aux conditions peuvent s'exprimer sous la forme d'expressions ou de fonctions similaires et que la sélection se fait sur une valeur simple. Au lieu de procéder à N tests, on récupère directement une valeur correspondante et on l'utilise.

Un exemple avec des expressions qui sont de simples sélections de valeurs :

```
# Nom du jour de la semaine.
if n == 1:
    jour = "lundi"
elif n == 2:
    jour = "mardi"
elif n == 3:
    jour = "mercredi"
elif n == 4:
    jour = "jeudi"
elif n == 5:
    jour = "vendredi"
elif n == 6:
    jour = "samedi"
elif n == 7:
    jour = "dimanche"
else:
    raise ValueError("jour invalide")
```

Qui peut aisément être remplacé par l'utilisation d'un dictionnaire (que l'on définira une fois pour toutes) :

```
joursem = { 1: "lundi", 2: "mardi", 3: "mercredi", 4: "jeudi",
            5: "vendredi", 6: "samedi", 7: "dimanche" }
jour = joursem[n]
```

Un exemple plus complexe où les expressions doivent pouvoir être paramétrées :

```
if forme == "cone":
    volume = 1/3 * pi * r ** 2 * h
elif forme == "cylindre":
    volume = pi * r ** 2 * h
elif forme == "sphere":
    volume = 4/3 * pi * r ** 3
```

Qui nécessite un peu plus de travail pour être remplacé par l'utilisation d'un dictionnaire :

```
def volcone(r, h):
    return 1/3 * pi * r ** 2 * h
def volcyl(r, h):
    return pi * r ** 2 * h
def volsph(r): # h inutilisé
    return 4/3 * pi * r ** 3
calcvol = { "cone": volcone, "cylindre": volcyl,
            "sphere": lambda r,h: volsph(r) # adaptateur pour la sphère
          }
volume = calcvol[forme](r, h)
```

Les blocs d'instructions sont définis une fois pour toutes dans des fonctions séparées. Un dictionnaire fait correspondre la fonction désirée pour chaque forme. À l'exécution on récupère simplement dans le dictionnaire la fonction à appeler suivant la forme, et on l'appelle dans la foulée avec les paramètres.

Pour les fonctions qui n'ont pas exactement les mêmes paramètres (nombre, ordre), on peut créer des adaptateurs en utilisant des fonctions anonymes (`lambda`) ou le wrapper de fonction `partial()` du module `functools`, ou encore profiter de l'existence de valeurs par défaut pour certains paramètres. Ces outils permettent de ramener les fonctions à un cas d'appel identique.

8.h - Fonction

L'instruction `def nomfonction(parametres):` permet de définir une nouvelle fonction avec son bloc d'instructions immédiatement derrière, indenté par rapport au `def`.

```
def afficher(lstnoms):
    print("il y a", len(lstnoms), "noms dans la liste:")
```

```
    for nom in lstnoms:
        print("\t-", nom.title())
```

La *sortie de la fonction* se fait à la fin du bloc d'instructions, ou lors de l'exécution d'une instruction **return** dans ce bloc. La levée d'une *exception*, si elle n'est pas stoppée à l'intérieur du corps de la fonction, provoque aussi la sortie de cette fonction.

Il n'y a pas de spécification de type pour les paramètres attendus par une fonction, ni pour la valeur retournée. Il n'y a pas même d'indication si la fonction retourne ou non quelque chose (pas de distinction fonction vs procédure, une « procédure » retournant toujours au moins **None**). On peut toutefois ajouter des attributs associés aux paramètres et à la valeur de retour de la fonction (voir *Annotations*, page 43).

Retour de valeur

Si la fonction se termine par une sortie de son bloc d'instructions sans rencontrer d'instruction **return**, ou si l'instruction **return** est utilisée seule, la fonction retourne la valeur par défaut **None**.

```
>>> def f(x):
...     y = x + 1
...
>>> res = f(3)
>>> print(res)
None
```

```
>>> def h(x):
...     z = x+2
...     return
...
>>> res = h(4)
>>> print(res)
None
```

L'instruction **return** permet de spécifier une valeur qui est retournée à l'appelant comme résultat de l'évaluation de la fonction.

```
>>> def g(x):
...     return x+1
...
```

```
>>> g(3)
4
```

En cas de retour de plusieurs valeurs, c'est en fait une séquence **tuple** qui est fabriqué via la notation séparant les valeurs par des virgules, et qui encapsule les différentes valeurs retournées.

```
>>> def diviseurmodulo(a, b):
...     diviseur = a // b
...     modulo = a % b
...     return diviseur, modulo
...
>>> diviseurmodulo(27,4)
(6, 3)
```

```
>>> d, m = diviseurmodulo(85,9)
>>> d
9
>>> m
4
```

Erreur courante : un oubli de **return** avec une valeur résultat produit une valeur **None**, par ailleurs certaines méthodes qui modifient l'objet auquel elles sont appliquées retournent **None** pour indiquer clairement que la modification a porté sur l'objet cible de la méthode. Ceci peut introduire des valeurs **None** dans les traitements, ce qui produit des erreurs comme :

```
AttributeError: 'NoneType' object has no attribute 'sort'
```

```
TypeError: unsupported operand type(s) for +: 'NoneType' and
'int'
```

Passage d'arguments

Lors de l'appel à la fonction, celle-ci reçoit dans ses paramètres des références vers les arguments fournis, avec les implications du **passage par référence** (voir *Objets mutables*, page 28).

Les arguments sont associés aux paramètres définis dans l'ordre d'apparition. On peut spécifier les noms des paramètres pour les argument et ainsi les fournir dans l'ordre de son choix, mais il faut alors le faire pour tous les suivants.

```
>>> def f(a, b, c):
...     return a + b + c
...
>>> f(1,2,3)
6
```

```
>>> f(1,c=3,b=2)
6
>>> f(b=2,a=1,c=3)
6
```

Il est possible d'utiliser directement les *valeurs contenues dans une séquence* en les faisant correspondre aux paramètres dans l'ordre, en utilisant la notation *****

```
>>> args  = (8, 2, 9)
>>> f(*args)
19
```

```
>>> args = (1, 4)
>>> f(8, *args)
13
```

Et il est possible d'utiliser directement les *valeurs contenues dans un dictionnaire* en les faisant correspondre aux paramètres via leurs noms, en utilisant la notation ******.

```
>>> params = {'a':1, 'b':5, 'c':2}
>>> f(**params)
8
```

```
>>> params = {'b':5, 'c':2}
>>> f(2, **params)
9
```

Valeur par défaut des paramètres

Il est possible d'indiquer des valeurs par défaut pour les paramètres lors de la définition de la fonction. Ces paramètres peuvent alors être omis lors de l'appel, et c'est la valeur par défaut qui est utilisée. Dès qu'un paramètre s'est vu associé une valeur par défaut dans la définition de la fonction, tous ceux qui le suivent doivent en avoir.

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

```
>>> f(1,c=4)
7
>>> f(5,3)
11
```

> **Attention** : la valeur spécifiée comme *valeur par défaut* pour un paramètre est *stockée lors de la définition de la fonction puis réutilisée* à chaque fois que ce paramètre est manquant dans un appel.
>
> Il n'est pas créé de nouvel objet, c'est toujours le même qui est réutilisé. Pas de problème pour les valeurs de types immutables, mais des effets de bords possibles pour les valeurs de types mutables (voir *Objets mutables*, page 28). Pour ces derniers, on préfère souvent utiliser une valeur par défaut à `None`, et tester cette valeur au début de la fonction pour mettre en place une *nouvelle* valeur mutable si besoin. Exemple : `if param is None: param = []`

Cette caractéristique d'effet mémoire dans les valeurs par défaut mutables a parfois été utilisée, comme dans l'exemple ci-dessous, où une fonction y accumule la somme des valeurs qu'on lui fournit lors de ses appels :

```
>>> def cumul(a, total=[0]):
...     total[0] += a
...     return total[0]
...
>>> cumul(3)
3
```

```
>>> cumul(4)
7
>>> cumul(12)
19
>>> cumul(1)
20
```

Dans les versions actuelles de Python, on préférera plutôt utiliser une *fermeture* où une fonction capture un contexte lors de sa définition et peut y stocker des valeurs (voir *Fermeture*, page 43).

Nombre variable d'arguments

Il est possible de définir une fonction acceptant un nombre variable d'arguments en utilisant les paramètres spéciaux **`*args`** et/ou **`**kwargs`** à la fin de la définition des paramètres (ce qui importe ici, c'est **`*`** et **`**`**) — on retrouve les mêmes notations que lors de l'appel des fonctions. Le premier reçoit, sous la forme d'un tuple, les arguments anonymes qui excèdent le nombre de paramètres. Le second reçoit, sous la forme d'un dictionnaire, les arguments nommés qui ne sont pas dans les paramètres.

Un exemple de réécriture de la fonction **`sum()`** avec un nombre variable d'arguments après une valeur initiale obligatoire.

```
>>> def somme(initial, *autres):
...     return initial + sum(autres)
...
```

```
>>> somme(10, 11, 12, 13)
46
>>> somme(4)
4
```

Un exemple d'utilisation d'un nombre variable d'arguments nommés pour afficher un menu.

```
>>> def choix(label, **options):
...     print(label)
...     for c in sorted(options):
...         print(c, options[c])
...     saisie = input("Choix:")
...     return saisie
...
>>> v = choix("Menu du jour", m1="Basique", m2="Travaillé", m3="Exceptionnel")
Menu du jour
m1 Basique
m2 Travaillé
m3 Exceptionnel
Choix:m2
>>> v
'm2'
```

L'utilisation de * et ** permet d'écrire facilement des adaptateurs (« wrappers ») génériques qui encapsulent l'appel de fonctions en ajoutant des fonctionnalités avant/après l'appel.

```
>>> from math import fmod
>>> def fctwrapper(f):
...     def wrapper(*args, **kwargs):
...         print("Appel avec", args, kwargs)
...         res = f(*args, **kwargs)
...         print("Retour:", res)
...         return res
...     return wrapper
...
>>> w = fctwrapper(fmod)
>>> w(5,2)
Appel avec (5, 2) {}
Retour: 1.0
1.0
```

Pour ce genre d'utilisation, les **décorateurs** sont particulièrement adaptés (voir *Décorateurs*, page 42).

> **Note** : on peut, lors de la définition, mettre le **`*`** avant les arguments nommés, cela force alors à utiliser les arguments nommés lors de l'appel de la fonction.

Fonction expression anonyme

Lorsqu'il y a besoin d'une fonction juste pour évaluer une expression (appeler une autre fonction en adaptant certains paramètres, faire un calcul, etc), on peut utiliser les fonctions anonymes créées avec le mot clé **`lambda`**. Une telle fonction peut prendre plusieurs paramètres, et ne contient qu'une expression dont la valeur est retournée.

```
>>> add = lambda x: x+1
>>> add(3)
4
>>> add(8)
9
```

Ces fonctions expressions anonymes sont couramment utilisées là où on doit fournir une fonction simple, comme par exemple avec la fonction standard builtin `filter()`, qui attend comme premier paramètre une fonction.

```
>>> list(filter(lambda x:x>0, [7,8,-2,0,7,-1,9]))
[7, 8, 7, 9]
```

On bénéficie avec les fonctions expressions anonymes de la fermeture (« closure ») comme détaillée ci-après (voir *Fermeture*, page 43).

```
>>> def increment(n):
...     return lambda x:x+n
...
>>> inc3 = increment(3)
>>> inc3(3)
6
>>> inc3(12)
15
```

Remplissage partiel de paramètres

Le module `functools` fournit une fonction `partial()` qui permet de créer un adaptateur à une fonction, pour laquelle une partie des arguments est déjà fournie.

```
>>> import functools
>>> def f(a,b,c,d):
...     print(a,b,c,d)
...
>>> p1 = functools.partial(f,c=4,d=7)
>>> p1(1,3)
1 3 4 7
>>> p2 = functools.partial(f, 8,9)
>>> p2(1,2)
8 9 1 2
```

Ceci permet de figer certains paramètres de fonctions et de créer ainsi des fonctions plus simples.

Généralisation de l'appel de fonctions

L'opération d'appel de fonction, qui est déclenché lorsqu'une parenthèse ouvrante suit un identificateur, est utilisable de façon générique :

- appliquée à un identificateur de fonction pour appeler celle-ci,
- appliquée à un identificateur de méthode d'un objet pour appeler celle-ci via l'objet auquel est appliquée la méthode,
- appliquée à un identificateur de classe pour créer un nouvel objet et l'initialiser,
- appliquée à un identificateur d'un objet possédant une méthode `__call__()` qui est appelée.

> **Définition** : tout objet auquel on peut appliquer l'appel fonctionnel avec des parenthèses est appelé un « **callable** ».
>
> La fonction standard builtin `callable()` permet de tester si un objet a cette propriété.

Décorateurs

Ce sont des fonctions (callable) chargées de transformer d'autres fonctions (ou méthodes, ou classes) lors de leur définition afin d'apporter de façon standardisée des services supplémentaires (contrôles en entrée/sortie, caches, traces, etc). Ils utilisent une syntaxe particulière[29] avec le symbole `@` (qui est une facilité d'écriture et une indication lors de la lecture du code).

29 Les décorateurs pourraient être simplement appelés après la définition de la fonction, en redéfinissant celle-ci, voir le PEP n° 318 « Decorators for Functions and Methods » : http://www.python.org/dev/peps/pep-0318/

Les décorateurs peuvent soit transformer, enrichir ce qu'ils décorent, soit carrément le masquer dans un autre objet d'adaptation, un « wrapper ». Le module standard `functools` fournit des fonctions `updatewrapper()` et `wraps()` afin de faciliter l'écriture de wrappers qui reprennent les méta-données des fonctions wrappées.

TODO

Ajouter un exemple simple et un exemple avec qq chose de plus complet.

Annotations

TODO

http://www.python.org/dev/peps/pep-3107/

http://stackoverflow.com/questions/3 038 033/what-are-good-uses-for-python3s-function-annotations

http://www.fightingquaker.com/pyanno/

Une fois les annotations en place, en les combinant avec des décorateurs associés aux fonctions ou méthodes, certains outils permettent de réaliser des contrôles lors de l'exécution, qui peuvent aller bien plus loin que le simple contrôle de type.

Voir par exemple la recette « Method signature type checking decorator for Python 3 »[30] dont voici des idées d'utilisation issues de la documentation :

```
@typecheck
def foo(i: int) -> bool:
    return a > 0

@typecheck
def to_int(*, s: by_regex("^[0-9]+$")) -> int:
    return int(s)

@typecheck
def set_debug_level(self, level: optional(one_of(1, 2, 3)) = 2):
    self._level = level

is_even = lambda x: x % 2 == 0
@typecheck
def multiply_by_2(i: int) -> is_even:
    return i * 2
```

Fermeture

Une fonction peut être définie à l'intérieur d'une autre fonction, la définition étant refaite à chaque appel, et produisant une nouvelle « valeur » fonction. Un exemple de constructeur qui crée de nouvelles fonctions (ayant toujours le même code) :

```
>>> def constructeur():
...     print("Construction.")
...     def f(x):
...         print("X", x, "dans",
id(f))
...     return f
...
>>> f1 = constructeur()
Construction.
>>> f2 = constructeur()
Construction.
>>> id(f1)
42180128
>>> id(f2)
139945231971336
>>> f1(3)
X 3 dans 42180128
>>> f2(3)
X 3 dans 139945231971336
```

30 http://code.activestate.com/recipes/572161-method-signature-type-checking-decorator-for-pytho/

L'intérêt semble limité. Mais la fermeture (« closure ») permet à une fonction, lorsqu'elle est définie par l'exécution d'une autre fonction, de capturer le contexte de sa définition et y piocher des éléments. Dans l'exemple ci-dessous on utilise la valeur du paramètre `n` tel qu'il est au moment de la définition de `fct`.

```
>>> def ajout(n):
...     def fct(x):
...         return x + n
...     return fct
...
>>> a3 = ajout(3)
>>> a10 = ajout(10)
>>> a3(3)
6
>>> a3(9)
12
>>> a10(0)
10
>>> a10(10)
20
>>> a3(10)
13
```

Pour reprendre l'exemple donné dans la section sur les valeurs par défaut, voici la façon de faire la même chose avec une fermeture (en utilisant les règles d'affectation dans les espaces de noms - voir *Règles d'affectation*, page 23 , qui permettent de modifier des valeurs dans le contexte capturé) :

```
>>> def cumulard():
...     total = 0
...     def cumule(x):
...         nonlocal total
...         total += x
...         return total
...     return cumule
...
>>> cumul = cumulard()
>>> cumul(3)
3
>>> cumul(4)
7
>>> cumul(12)
19
>>> cumul(1)
20
```

Ce qui permet en plus de définir plusieurs fonctions de cumul ayant chacune son propre accumulateur.

8.i - Exceptions

TODO

8.j - Blocs de contexte géré

TODO

+ renvoi vers l'interface des objets définissant les contextes.

III - TYPES ET OPÉRATIONS

Nous allons faire ici un rapide tour des types standards disponibles en Python, des opérations réalisables et des syntaxes utilisées.

La fonction standard `type()` permet de connaître le type d'une expression à l'exécution, et la fonction standard `dir()` permet de connaître les opérations et attributs liés à l'espace de noms de l'objet (opérateurs compris).

1 - Numérique

Python dispose des deux types numériques habituels, entiers (`int`) et flottants (`float`), ainsi que d'un type standard pour les nombres complexes (`complex`). Le type entier est du genre « bigint », où la représentation n'est limitée que par l'espace mémoire, pas de risque de débordement ou de « rollover ». Le type flottant reprend le stockage des nombres du type double du langage C.

L'écriture littérale des entiers utilise ce que l'on trouve dans la plupart des langages. En plus de la représentation en base he**x**adécimale (préfixe `0x`), il y a une extension permettant de les représenter en base **b**inaire (préfixe `0b`) et une variation sur la représentation en base **o**ctale (`0o` où un petit o suit le zéro, le zéro seul en tête ne suffit pas et est considéré comme une erreur de syntaxe).

```
>>> 111
111
>>> 0x111
273
```

```
>>> 0o111
73
>>> 0b111
7
```

```
>>> -42
-42
>>> +23
23
```

Pour les nombres flottants, c'est la représentation littérale habituelle avec la notation décimale via un point (`.`) et la lettre `e` ou `E` pour introduire la puissance de 10.

```
>>> 1.
1.0
>>> .056
0.056
```

```
>>> -156e-6
-0.000156
>>> 1.3E8
130000000.0
```

```
>>> 0.000003
3e-06
>>> 1.9E24
1.9e+24
```

Pour les nombres complexes, une notation avec un suffixe `j` permet de regrouper deux valeurs composant la partie réelle et la partie imaginaire du nombre (exprimées et stockées chacune sous la forme d'un flottant). L'utilisation de parenthèses permet de regrouper les composants du nombre complexe dans une expression. Un nombre complexe possède deux attributs en lecture seule `.real` et `.imag` et une méthode `.conjugate()`.

```
>>> 8j
8j
>>> 4+5j
(4+5j)
>>> 4+(3j)*6
(4+18j)
```

```
>>> a = 5.2-3.2e5j
>>> a
(5.2-320000j)
>>> a.real
5.2
>>> a.imag
-320000.0
```

```
>>> a.conjugate()
(5.2+320000j)
```

L'affichage dans le shell Python se fait par défaut en décimal pour les nombres entiers, et dans un format adapté pour les nombres flottants. Les opérations de formatage de nombres seront vues lorsque les chaînes de caractères seront abordées.

1.a - Opérateurs et fonctions

Les opérations courantes d'arithmétique sont utilisables entre ces types, *addition* (`+`), *soustraction* (`-`), *multiplication* (`*`), *division* (`/`). On trouve aussi la *division entière* (`//`), le reste de division entière ou *modulo* (`%`), l'élévation à la *puissance* (`**`).

> **Attention** : en Python3[31], la division `/` *même entre deux entiers* est une division flottante, comme dans une expression de calcul en physique ou en mathématiques où la division de deux nombres $\frac{1}{2}$ vaut 0,5 et non 0. La division entière doit être explicitement appelée avec l'opérateur `//`.

Quelques fonctions mathématiques de base sont accessibles directement dans l'espace des noms standards builtin : la fonction de valeur absolue `abs()`, l'élévation à la puissance via l'appel d'une fonction `pow()`, le calcul du diviseur et du reste de la division entière avec `divmod()`, l'arrondi d'un nombre flottants avec `round()`.

Le module `math`[32] fournit les *fonctions mathématiques de la librairie C* et quelques fonctions supplémentaires : `acos()`, `acosh()`, `asin()`, `asinh()`, `atan()`, `atan2()`, `atanh()`, `ceil()`, `copysign()`, `cos()`, `cosh()`, `degrees()`, `erf()`, `erfc()`, `exp()`, `expm1()`, `fabs()`, `factorial()`, `floor()`, `fmod()`, `frexp()`, `fsum()`, `gamma()`, `hypot()`, `isfinite()`, `isinf()`, `isnan()`, `ldexp()`, `lgamma()`, `log()`, `log10()`, `log1p()`, `log2()`, `modf()`, `pow()`, `radians()`, `sin()`, `sinh()`, `sqrt()`, `tan()`, `tanh()`, `trunc()`, ainsi que les constantes `e` et `pi`.

On a les priorités habituelles dans les expressions arithmétiques[33], dans l'ordre :

1. les expressions parenthésées avec `(` et `)`,
2. l'appel de fonctions (avec l'évaluation des paramètres avant l'appel),
3. l'élévation à la puissance `**`[34],
4. les `+` `-` unaires,
5. les opérateurs multiplicatifs `*` `/` `//` `%`,
6. les opérateurs additifs binaires `+` `-`.

```
>>> (3+4)*5
35
>>> from math import *
>>> sqrt(sin(0.5)**2+cos(0.5)** 2)
1.0
>>> 2**-1
0.5
```

```
>>> (abs(-3)+3)/4
1.5
>>> cos(radians(90/2))
0.7071067811865476
>>> hypot(3,4)
5.0
```

1.b - Opérateurs de bits

Sur les nombres entiers, on dispose aussi des opérateurs de manipulations de bits, **décalage** à gauche (`<<`) et à droite (`>>`), **et** bit à bit (`&`), **ou** bit à bit (`|`), **ou exclusif** bit à bit (`^`) , opération unaire d'**inversion** des bits (`~`). Les exemples ci-dessous utilisent des valeurs littérales ainsi qu'une représentation du résultat exprimées en binaire pour plus de lisibilité.

```
>>> bin(0b010001 << 4)
'0b100010000'
>>> bin(0b010001 >> 2)
'0b100'
>>> bin(0b0101 & 0b1100)
'0b100'
```

```
>>> bin(0b0101 | 0b1100)
'0b1101'
>>> bin(0b0101 ^ 0b1100)
'0b1001'
>>> bin(0b1111 ^ 0b1010)
'0b101'
```

```
>>> bin(~0b0101)
'-0b110'
```

31 En Python2 l''opérateur de division / entre deux entiers donne un résultat entier : 1/2 vaut 0.
32 Pour le support des nombres complexes, les mêmes fonctions sont disponibles dans le module `cmath`.
33 Les priorités seront reprises plus loin en intégrant l'ensemble des expressions possibles dans le langage.
34 L'élévation à la puissance permet tout de même d'évaluer d'abord une valeur signée sur sa droite, 2**-1 vaut 0.5.

1.c - Transtypage vers numérique

Les types `int`, `float` et `complex` peuvent être utilisés avec une écriture fonctionnelle pour effectuer un transtypage. Ceci est entre autres utilisé avec les chaînes de caractères pour réaliser les conversions vers les numériques. Pour les entiers, lorsqu'on transtype une chaîne de caractères, il est possible de spécifier un second paramètre qui indique la *base* dans laquelle la valeur est exprimée. Pour les complexes, il est possible de spécifier deux paramètres : partie réelle et partie imaginaire. Les booléens `False` et `True` sont transtypés respectivement vers les nombres `0` et `1`.

```
>>> int(45.2)
45
>>> int(45.86)
45
>>> float(-12)
-12.0
>>> complex(22,8)
(22+8j)
```

```
>>> int("-6792")
-6792
>>> float("3.8e-2")
0.038
>>> int("10001", 2)
17
>>> int("27GH728IJ", 24)
255451295683
```

```
>>> complex("78+3j")
(78+3j)
>>> int(True)
1
>>> int(False)
0
>>> complex(False)
0j
```

Pour les conversions flottants vers entiers, voir aussi les fonctions `ceil()`, `floor()`, `trunc()` du module `math`.

Des méthodes spécifiques permettent de redéfinir les opérations de transtypage vers numérique (voir *Redéfinition des transtypages*, page 51).

1.d - Méthodes spéciales

Pour les entiers on dispose des méthodes `.bit_length()`, `.to_bytes()` et `.from_bytes()`. Et pour les flottants les méthodes `.as_integer_ratio()`, `.is_integer()`, `.hex()`, `.fromhex()`.

2 - Logique

Le type booléen (`bool`) accepte deux valeurs littérales, `True` et `False`. Il s'agit d'une sous-classe du type entier (`int`), ce qui permet d'utiliser l'un à la place de l'autre, avec une correspondance `True`⟺`1` et `False`⟺`0`.

> **Attention** : les fonctions et opérations logiques *dans les librairies* peuvent retourner un booléen `False`/`True` ou bien un entier `0`/`1`.

2.a - Transtypage vers booléen

Comme pour les nombres, l'utilisation du type `bool` avec une écriture fonctionnelle permet d'effectuer un transtypage. Pour les types de base les règles sont simples : les *valeurs nulles*, la constante `None` et les *conteneurs vides* correspondent à la valeur `False`, tout le reste correspond à la valeur `True`. Pour les valeurs d'autres types, elles correspondent à la valeur `True` à moins d'avoir redéfini l'opération de transtypage vers booléen ou l'opération de calcul de longueur (voir Redéfinition des opérateurs page 51).

```
>>> bool(None)
False
>>> bool(0)
False
>>> bool(1)
True
```

```
>>> bool("")
False
>>> bool("Coucou")
True
>>> bool([])
False
```

```
>>> bool([5, 2, 8])
True
>>> class C: pass
...
>>> o = C()
>>> bool(o)
True
```

2.b - Opérateurs logiques

Le langage supporte les opérateurs booléens binaires **et logique** (`and`) et **ou logique** (`or`), ainsi que l'opérateur unaire d'**inversion** (`not`).

```
>>> a = True
>>> not a
False
>>> a or False
True
>>> a and False
False
```

```
>>> b = not a or a
>>> b
True
>>> b and a or not b and not a
True
```

Python pratique l'**évaluation au plus court** (« shortcut evaluation ») ; dès que la valeur finale d'une expression est connue, l'évaluation s'arrête. Ainsi l'évaluation d'un *et logique* s'arrête dès qu'un opérande faux est rencontré (la valeur finale est `False` de toutes façons), et l'évaluation d'un *ou logique* s'arrête dès qu'un opérande vrai est rencontré (la valeur finale est `True` de toutes façons).

> **Attention** : les opérateurs `and` et `or` *retournent toujours un de leurs opérandes* (celui qui a décidé en dernier, par son transtypage en booléen, de la valeur de l'expression lors de l'évaluation au plus court).
>
> Pour être sûr d'avoir une valeur booléenne résultant d'une expression, il faut s'assurer que tous les opérandes de l'expression sont typés booléens — ou par sécurité faire un transtypage explicite.

```
>>> [] or False or "Non vide" or True
'Non vide'
>>> bool([] or False or "Non vide" or True)
True
>>> ["qqchose"] and 42 and 0.0 and True
0.0
>>> bool(["qqchose"] and 42 and 0.0 and True)
False
```

Ceci est parfois utilisé comme écriture condensée pour n'exécuter du code que dans certaines conditions. Dans l'exemple ci-dessous la fonction *sinus* n'est appelée que lorsque l'angle est supérieur à $\frac{\pi}{2}$.

```
>>> angle = 2.523
>>> (angle > pi / 2) and sin(angle)
0.5798891765851072
```

```
>>> angle = 0.45
>>> (angle > pi / 2) and sin(angle)
False
```

On trouve de temps en temps ce genre d'écriture dans le code, mais on préfère autant que possible utiliser soit une instruction de test (voir *Test*, page 30), soit une expression conditionnelle explicite (voir *Expression conditionnelle*, page 31) ; deux écritures plus lisibles et compréhensibles.

2.c - Opérations de comparaison

Toutes les opérations de comparaison entre deux valeurs[35] sont des opérations qui *produisent des valeurs booléennes*. On dispose des comparateurs habituels, **plus grand** (`>`), **plus grand ou égal** (`>=`), **plus petit** (`<`), **plus petit ou égal** (`<=`), **égal** (`==`) et **différent** (`!=`).

2.d - Comparaison d'identité

On dispose aussi des opérateurs de comparaison sur l'**identité des objets**, `is` et `is not`.

```
>>> v1 = 3.14159
>>> v2 = v1
>>> v3 = 3.14159
```

```
>>> v2 == v1
True
```

35 Pour les comparaisons plus grand/plus petit, il faut des valeurs ayant une relation d'ordre.

```
>>> v2 is v1
True
>>> v3 == v1
True
```

```
>>> v3 is v1
False
```

> **Note** : concernant l'*identité des entiers et chaînes de caractères*, Python réalise automatiquement une opération de mise en cache (cf fonction `intern()` du module `sys`) pour les chaînes courtes[36] et pour les nombres entiers courants (faibles valeurs), qui consiste à réutiliser toujours le même objet pour ces valeurs (on parle de « interned values »).

```
>>> a = "A"
>>> c = "A"
>>> a is c
True
>>> n1 = 42
>>> n2 = 42
>>> n1 is n2
True
```

```
>>> for i in range(1000):
...     n = i+1
...     if n is not i+1:
...         print(n, "non caché")
...         break
...
257 non caché
```

Cela évite de multiplier les objets en mémoire, limite les opérations de création/destruction, et accélère les opérations de test d'égalité (test sur l'identité avant d'appeler les opérateurs de comparaison de plus haut niveau), ainsi que les opérations de recherche dans les tables de hachage.

Les chaînes ainsi mises en cache sont tout de même détruites lorsqu'elles ne sont plus référencées ; par contre les entiers mis en cache sont créés au démarrage de Python et perdurent tout au long de l'exécution.

Identité et durée de vie

> **Attention** : en cpython l'identité étant l'adresse de l'objet en mémoire, lorsqu'un objet est détruit cette mémoire est à nouveau disponible et peut donc être réutilisée pour un nouvel objet qui aura alors la même identité.

Deux objets qui existent simultanément temporellement ne peuvent donc pas avoir la même identité. Par contre, deux objets qui existent dans des instants différents peuvent se retrouver avec la même identité (adresse) en raison de la réutilisation de la mémoire libérée.

Dans l'exemple ci-dessous, a et b existent en même temps, leurs *ids* sont différents. Par contre, le second test d'égalité sur les identités de *deux objets créés et détruits immédiatement* (comptage de références) montre que l'adresse mémoire a été réutilisée lors de l'évaluation de l'expression.

```
>>> class A: pass
...
>>> class B: pass
...
>>> a = A()
```

```
>>> b = B()
>>> id(a) == id(b)
False
>>> id(A()) == id(B())
True
```

Comparaison avec None

Pour tester si une valeur vaut le **singleton** `None`, on utilise le *test sur l'identité*.

```
>>> a = None
>>> a is None
True
```

```
>>> b = 4
>>> b is not None
True
```

36 Ainsi que pour tous les identificateurs utilisés.

2.e - Test de valeur booléenne

Lorsqu'un booléen est nécessaire dans une expression, soit que l'instruction attend explicitement une expression booléenne (le test `if`, la boucle conditionnelle `while`), soit que l'on utilise des opérateurs logiques, un transtypage est automatiquement réalisé.

D'une façon générale, lorsque l'on a besoin de tester une valeur booléenne, on ne teste pas l'égalité ou l'identité avec `True` ou `False`, mais **on laisse le langage faire un transtypage** vers `bool`.

Dans les exemples suivant, on vide une liste par la fin en affichant les valeurs au fur et à mesure, et on fait saisir à nouveau un nombre entier tant que la valeur fournie est nulle.

```
>>> lst=["boum!","un","deux","trois"]
>>> while lst:
...     print(lst.pop())
...
trois
deux
un
boum!
>>> lst
[]
```

```
>>> x = 0
>>> while not x:
...     x = int(input("Valeur:"))
...
Valeur:0
Valeur:0
Valeur:3
>>> x
3
```

3 - Conteneurs

3.a - Opérations génériques

len sum max min

indexation

filter / map / reduce, voir si c'est utilisable avec n'importe quel itérable (en tout cas, filter et map retournent des itérables spécifiques)

? Programmation fonctionnelle

map /reduce / filter

Voir http://ua.pycon.org/static/talks/kachayev/#/28

3.b - Chaînes de caractères

3.c - Tuples

3.d - Listes

Listes en compréhension

Générateurs en compréhension

Lorsque l'expression en compréhension est mise entre parenthèses, et on entre crochets,

3.e - Dictionnaires

Dictionnaires en compréhension

+ info salt dans les clés de hachage pour éviter les attaques par collisions, renforce le côté non déterministe de l'ordre.

3.f - Ensembles

Ensembles en compréhension

Ensembles immutables

3.g - Module collections

Tuples nommés

namedtuple

Files à "double extrémité"

dequeue

Compteurs

Counter

Dictionnaires ordonnés

OrderedDict

Dictionnaires avec valeur par défaut

defaultdict

4 - Système de classes

4.a - Redéfinition des opérateurs

4.b - Redéfinition des transtypages

4.c - Redéfinition de l'affectation augmentée

4.d - Redéfinition des séquences

4.e - Redéfinition de l'itération

Cas particulier des générateurs avec throw().

4.f - Redéfinition de l'appel de fonction

__call__ + re-indication de l'utilisation d'une classe comme une fonction pour créer de nouvelles instances.

TODO : Voir "functor", si c'est défini dans la doc python ou dans la littérature

4.g - Contrôle d'accès aux attributs

TODO : les property et accesseurs

4.h - Classes abstraites

TODO : renvoi vers l'article de linuxmag.

IV - USAGES COURANTS

1 - Entrées / sorties

1.a - Console

TODO : print et input

1.b - Fichiers texte

TODO : open, méthodes, print(file=)

1.c - Fichiers binaires (?)

TODO : est-ce nécessaire ? peut-être directement le bas niveau os.open() ?

2 - Manipulations XML

3 - Protocoles de l'Internet

3.a - HTTP

4 - Manipulations bas niveau

TODO : Modules array, struct & Co.

4.a - Appel direct de code C/C++

TODO : Modules ctypes

V - ANNEXES

1 - Options Python en ligne de commande

TODO : Les options spécifiques à Python, mettre ailleurs (éventuellement le sys.argv et les modules de gestion des options cli)

2 - Python2 vs Python3

Module six pour faciliter le code portable.

- print and exec are now functions, not statements
- the dangerous input function is gone
- raw_input is renamed input
- the distinction between int and long is gone
- the second-string "classic classes" are gone
- strings are Unicode, not bytes
- division is done correctly
- the error-prone syntax of the "except" statement is fixed
- inconsistently named modules are fixed
- the plethora of similar dict methods is cleaned up
- map, filter, zip operate lazily rather than eagerl

Voir éventuellement si des choses à reprendre chez Django (expériences...) https://docs.django-project.com/en/dev/topics/python3/

Et chez pocoo http://lucumr.pocoo.org/2011/1/22/forwards-compatible-python/

Et http://python3porting.com/noconv.html

2.a - Entiers

En Python2 deux types entiers cohabitent, les int sur 32 ou 64 bits suivant la machine et les long sans limitation de taille. Lors de l'évaluation d'une expression, en cas de dépassement de capacité des entiers, le passage aux entiers longs est automatique.

TODO : Entier normal et long en Python2, avec la conversion automatique, un seul type d'entier en Python3.

2.b - Chaînes de caractères

TODO : Distinction plus claire en Python 3 str vs bytes. Mix malheureux en Python2.

Préfixe u"" présent dans Python2, et réintroduit dans Python3.3 (absent en 3.0 à 3.2), pour faciliter l'écriture de code portable.

2.c - Affichage avec print

TODO : Builtin intern() [python2] maintenant dans le module sys [python3]

2.d - Ouverture de fichiers et encodage

TODO : Fct standard open() [python3] vs utilisation codecs.open()[python2] pour pouvoir spécifier l'encodage.

2.e - Itérateurs et next

Utilisation de la fonction builtin `next()` pour être portable car dans Python2 les itérateurs fournissent la valeur suivante via la méthode utilisent `.next()` et dans Python3 l'interface a été homogénéisé et ils utilisent la méthode `.__next__()`.

- cf PEP3114 http://www.python.org/dev/peps/pep-3114/)

2.f - Héritage et super

En Python2 super() ne fonctionne que pour les « new style class », celles qui héritent de object.

3 - Documentation du code

```
def troubillage(cogn, nbfois):
    """Troubille le cognoscope, retourne son état.

    :param cogn: cognoscope à troubiller.
    :type cogn: Cognoscope
    :param nbfois: nombre de fois où il faut troubiller.
    :type nbfois: int
    :return: indicateur du nouvel état du cognoscope après troubillage.
    :rtype: str
    """
    …
```