Python

Précis et concis

Python 3.4 & 2.7

Mark Lutz

Traduit de l'anglais par
Dominique Maniez

DUNOD

© 2017 Dunod
Authorized French translation of material from the English edition of Python Pocket Reference, 5th Edition
ISBN 9781449357016 © 2014 Mark Lutz

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

Conception de la couverture : Randy Comer
Réalisation de la couverture française :
Pierre-André Gualino

© Dunod, 2017
11 rue Paul Bert, 92240 Malakoff
www.dunod.com
ISBN 978-2-10-075945-3

Table des matières

1

Python précis et concis

INTRODUCTION

Python est un langage de programmation générique, open source, prenant en charge plusieurs modèles de programmation (procédural, fonctionnel et orienté objet). On l'utilise aussi bien pour créer des programmes autonomes que des scripts dans une grande variété de domaines, et on estime que c'est l'un des langages de programmation les plus utilisés au monde.

La lisibilité du code, la fonctionnalité de ses librairies[1], une conception qui optimise la productivité du développeur, la qualité logicielle, la portabilité des programmes, et l'intégration des composants sont les principales caractéristiques de Python. Python peut

1. Note du traducteur : dans cet ouvrage, nous avons délibérement choisi l'anglicisme librairie à la place de bibliothèque car nous avons constaté l'emploi croissant de ce terme chez les informaticiens, même s'il peut être critiqué. De la même manière, nous avons préféré assignation à affectation, package à paquetage, etc. car nous pensons que dans une traduction technique c'est l'usage en vigueur chez les professionnels qui doit primer. Pour certains termes très techniques, nous avons préféré garder l'anglais ou bien nous avons mentionné le terme original entre parenthèses après sa traduction française.

tourner sur la plupart des systèmes d'exploitation actuels tels que Unix, Linux, Windows, Macintosh, Java, .NET, Android ou iOS, pour ne citer que ceux-là.

Le présent ouvrage présente de façon synthétique les types et les instructions Python, les noms des méthodes spéciales, les fonctions intégrées et les exceptions, ainsi que les modules de librairies standards couramment utilisés et d'autres outils Python dignes d'intérêt. Les développeurs en feront leur manuel de référence qui viendra compléter d'autres publications contenant des tutoriels, des exemples de code et d'autres ressources d'apprentissage.

Cette *cinquième édition* traite de Python 3.X et 2.X. S'il est vrai qu'elle est centrée sur la version 3.X, elle couvre néanmoins les différences de la version 2.X. Cette édition a été mise à jour pour être en phase avec les versions *3.3* et *2.7*, mais aussi avec les améliorations de la version *3.4*. Toutefois, l'essentiel reste applicable à l'ensemble des versions 2.X et 3.X, qu'elles soient antérieures ou ultérieures.

Cette édition s'applique également à toutes les grandes implémentations de Python, notamment CPython, PyPy, Jython, IronPython, et Stackless. Elle a également été mise à jour et complétée pour suivre les évolutions du langage, des librairies et des pratiques. Parmi ces nouveautés on trouvera : MRO et les algorithmes formels d'héritage de `super()`, les imports, les gestionnaires de contextes, l'indentation des blocs, ainsi que des modules de librairies tels que `json`, `timeit`, `random`, `subprocess`, `enum`, et le nouveau lanceur Windows.

CONVENTIONS TYPOGRAPHIQUES

Le présent ouvrage adopte les conventions suivantes :

`[]`
: Dans la syntaxe des instructions, les éléments entre crochets sont optionnels, mais ils sont parfois utilisés littéralement (par exemple, dans les listes dont les éléments sont encadrés par des crochets). Ceci est alors précisé dans le texte.

`*`
: Dans la syntaxe des instructions, les éléments suivis d'un astérisque peuvent être répétés. L'étoile est parfois utilisée de manière littérale dans Python (par exemple, la multiplication).

`a | b`
: Dans la syntaxe des instructions, les éléments séparés par une barre verticale représentent des alternatives ; la barre est parfois utilisée de manière littérale dans Python (par exemple, l'union).

Italique
: Utilisé pour les noms de fichiers et les URL, ou pour mettre en évidence des termes nouveaux ou importants.

`Police à chasse fixe`
: Utilisée pour le code, les commandes et les options de lignes de commande ; sert également à indiquer les noms de modules, de fonctions, d'attributs, de variables et de méthodes.

`Police à chasse fixe en italique`
: Utilisée pour les noms des paramètres remplaçables au sein d'une ligne de commande, d'une expression, d'une fonction ou d'une méthode.

`Fonction()`
: Sauf indication contraire, les fonctions et méthodes susceptibles d'être invoquées sont suffixées d'une paire de parenthèses pour les différencier d'autres types d'attributs.

Voir *Nom de section*
: Les références aux autres sections de cet ouvrage sont indiquées par des titres de sections en italique.

Note

Dans ce livre, la mention « 3.X » et « 2.X » signifie que le sujet traité s'applique à toutes les versions de Python couramment utilisées. Quand un numéro précis de version est spécifié, cela indique une portée plus limitée (par exemple, « 2.7 » signifie que cela ne s'applique qu'à la version 2.7). Dans la mesure où de futures modifications du langage Python peuvent invalider le contenu de cet ouvrage, il est prudent de se reporter aux notes de version « What's New » qui renseignent sur les nouveautés des versions de Python publiées après la sortie de ce livre et que l'on peut trouver en ligne à :

http://docs.python.org/3/whatsnew/index.html

UTILISATION DE LA LIGNE DE COMMANDE PYTHON

Les lignes de commande pour exécuter les programmes Python à partir du shell ont le format suivant :

```
python [option*]
  [fichier de script| -c commande | -m module | - ] [arg*]
```

Dans cette syntaxe, `python` désigne l'interpréteur exécutable Python, avec son chemin complet, ou bien avec le mot `python` résolu par le shell (par exemple, via le paramètre `PATH`). Les options pour Python lui-même apparaissent avant le nom du programme à exécuter (*`option`*). Les arguments destinés au code à exécuter n'apparaissent qu'après le nom du programme (*`arg`*).

Options de la commande Python

Voici en Python 3.X (voir plus bas la section intitulée *Options de la commande Python 2.X* pour les différences) les éléments *`option`* de la commande Python qui sont utilisés par Python lui-même :

`-b`
: Émet un avertissement en cas d'appel de `str()` avec un objet `bytes` ou `bytearray` sans argument d'encodage, et en cas de comparaison d'un objet `bytes` ou `bytearray` avec un objet `str`. L'option `-bb` provoque à la place des erreurs.

`-B`
: N'écrit pas les fichiers de code intermédiaire (bytecode) `.pyc` ou `.pyo` lors de l'import des modules source.

`-d`
: Active la sortie de l'analyseur de débogage (pour les développeurs du noyau de Python).

`-E`
: Ignore les variables d'environnement Python (comme `PYTHONPATH` qui sont décrites plus bas).

`-h`
: Affiche le message d'aide et termine le programme.

`-i`
: Passe en mode interactif après l'exécution d'un script. Conseil : cette option est utile pour le débogage après une exception ;

voir aussi `pdb.pm()`, qui est décrit dans la documentation de la librairie Python.

`-O`

Optimise le code intermédiaire généré (crée et utilise des fichiers bytecode `.pyo`). N'apporte, à cette heure, qu'une amélioration mineure des performances.

`-OO`

Fonctionne comme l'option précédente `-O`, mais supprime également la documentation (docstrings) du code intermédiaire.

`-q`

N'affiche pas le numéro de version ni le message de copyright lors du démarrage en mode interactif (à partir de Python 3.2).

`-s`

N'ajoute pas le répertoire du site de l'utilisateur au chemin de recherche du module `sys.path`.

`-S`

Ne réalise pas d'import du module `site` à l'initialisation.

`-u`

Force `stdout` et `stderr` à passer en mode binaire et sans tampon (unbuffered).

`-v`

Affiche un message chaque fois qu'un module est initialisé, en indiquant l'emplacement à partir duquel il a été chargé ; en répétant le paramètre (`-vv`), on obtient davantage d'information.

`-V`

Affiche le numéro de version de Python et quitte le programme (identique à `--version`).

`-W` *arg*

Contrôle les avertissements : *arg* prend la forme action:message:catégorie:module:numéro_de_ligne. Voir aussi les sections *Avertissements* et *Exceptions de catégories d'avertissements*, ainsi que la documentation du module des avertissements dans le manuel de référence de la librairie Python (http://www.python.org/doc/).

`-x`

Saute la première ligne du code source, ce qui permet l'utilisation de formes `#!cmd` non compatibles avec Unix.

`-X` *`option`*
: Définit une option spécifique à une implémentation de Python (à partir de Python 3.2) ; voir la documentation spécifique de l'implémentation pour les valeurs de l'*`option`*.

Spécifications du programme à exécuter en ligne de commande

Le code à exécuter et les arguments de la ligne de commande qui sont passés en paramètre sont spécifiés de la manière suivante dans la ligne de commande Python :

`Fichier_de_script`
: Spécifie le nom du script Python à exécuter qui représente le fichier principal du programme (par exemple, *`python`* `main.py` exécute le code de `main.py`). Le nom du script, qui peut être un chemin absolu ou relatif (désigné par « . »), est accessible dans `sys.argv[0]`. Avec certaines plateformes, les lignes de commandes peuvent aussi omettre l'élément *`python`* si elles commencent par un nom de script et si elles ne contiennent pas d'options pour Python.

`-c` *`commande`*
: Spécifie quel code Python (sous la forme d'une chaîne de caractères) doit être exécuté (par exemple : *`python`* `-c "print('spam' * 8)"` demande à Python d'effectuer une opération d'affichage). `sys.argv[0]` renvoie la valeur `'-c'`.

`-m` *`module`*
: Lance un module en tant que script : recherche un *`module`* dans `sys.path` et l'exécute en tant que fichier racine (par exemple : *`python`* `-m pdb s.py` exécute le module de débogage `pdb` situé dans un répertoire de librairies standard, avec l'argument `s.py`). Le *`module`* peut également désigner un package (par exemple :`idlelib.idle`). `sys.argv[0]` renvoie le chemin complet du module.

  Lit les commandes Python en provenance du flux d'entrée standard, `stdin` (par défaut) ; passe en mode interactif si `stdin` est un « tty » (terminal interactif). `sys.argv[0]` renvoie `''`.

*arg**
: Signale que tout autre élément de la ligne de commande est passé au fichier de script ou à la commande, et apparaît dans la liste des chaînes `sys.argv[1:]`.

Si aucun *`fichier_de_script`*, *`commande`* ni *`module`* n'est spécifié, Python entre en mode interactif et lit les commandes depuis `stdin` (en utilisant `readline` pour l'entrée s'il est installé), `sys.argv[0]` renvoyant une chaîne vide (`''`), sauf si l'option – a été employée.

Parallèlement aux lignes de commandes traditionnelles exécutées dans un shell, il est en général possible de lancer des programmes Python en cliquant sur leur nom de fichier depuis un explorateur de fichiers dans une interface graphique, en appelant des fonctions de la librairie standard Python (par exemple : `os.popen()`), et en utilisant des options de menus de lancement des programmes dans des IDE tels que IDLE, Komodo, Eclipse et NetBeans.

Options de la commande Python 2.X

Python 2.X accepte le même format de ligne de commande, mais ne prend en charge ni l'option `–b`, qui est liée aux changements de types de chaînes de Python 3.X, ni `–q`, ni non plus `–X`, qui sont des ajouts récents de la version 3.X. En revanche, il accepte d'autres options des versions 2.6 et 2.7 (certaines sont même présentes dans des versions antérieures) :

`-t` et `–tt`
: Émet des avertissements en cas de mélanges incohérents de tabulations et d'espaces dans les indentations. Si l'on emploie l'option `-tt`, les erreurs sont en revanche déclenchées. Python 3.X traite toujours ces incohérences comme des erreurs de syntaxe (voir la section intitulée *Règles syntaxiques*).

`-Q`
: Les options en rapport avec la division sont les suivantes : `-Qold` (par défaut), `-Qwarn`, `-Qwarnall` et `–Qnew`. Toutes sont supplantées par le nouveau comportement de la véritable division de Python 3.X (voir la section *Notes sur l'utilisation des opérateurs*).

`-3`

Émet des avertissements pour toute incompatibilité Python 3.X dans le code que l'utilitaire *2to3*, chargé de l'installation standard de Python, ne peut pas réparer facilement.

`-R`

Pour lutter contre les attaques par déni de service, les valeurs de hachage des différents types sont modifiées par un nombre pseudo-aléatoire, si bien qu'il est impossible de les deviner entre les invocations successives de l'interpréteur. Cette option est nouvelle à partir de Python 2.6.8. Ce commutateur est également présent dans la version 3.X à partir de la version 3.2.3 par souci de compatibilité, mais cette création aléatoire des valeurs de hachage est activée par défaut depuis la version 3.3.

VARIABLES D'ENVIRONNEMENT DE PYTHON

Les variables d'environnement (l'environnement est également appelé *shell*) sont des paramètres au niveau du système qui s'étendent à tous les programmes et qui sont utilisés pour la configuration globale.

Variables opérationnelles

Voici les principales variables d'environnement configurables par l'utilisateur et permettant la modification du comportement des scripts :

`PYTHONPATH`

Ajoute au chemin de recherche par défaut celui des fichiers de modules importés. Le format de la valeur de cette variable est identique au paramètre `PATH` du shell : chemins de répertoires séparés par le caractère deux points (point-virgule sous Windows). Si la variable est définie, les imports de modules lancent une recherche des fichiers importés dans chaque répertoire listé dans `PYTHONPATH`, de la gauche vers la droite. Ils sont ajoutés à `sys.path` (le chemin complet de recherche des modules pour les composants les plus à gauche des imports absolus) après le répertoire du script, et avant les répertoires

des librairies standards. Voir `sys.path` dans les sections *Module sys* et *Instruction import*.

`PYTHONSTARTUP`

Si l'on assigne un nom de fichier lisible à cette variable, les commandes Python de ce fichier sont exécutées avant que la première invite ne soit affichée en mode interactif (ce qui est pratique pour les outils d'usage fréquent).

`PYTHONHOME`

Si la variable est définie, sa valeur est utilisée comme répertoire alternatif pour les modules des librairies (ou `sys.prefix`, `sys.exec_prefix`). Le chemin par défaut de recherche des modules est : `sys.prefix/lib`.

`PYTHONCASEOK`

Si la variable est définie, Python ne fait pas la différence entre les minuscules et les majuscules dans les noms des fichiers des instructions d'imports (cela ne fonctionne actuellement que pour Windows et OS X).

`PYTHONIOENCODING`

La valeur de la variable est assignée à la chaîne *`encodingname`*[`:`*`errorhandler`*] pour remplacer l'encodage Unicode par défaut (et le gestionnaire d'erreurs optionnel) utilisé pour les transferts de texte vers les flux `stdin`, `stdout` et `stderr`. Ce paramètre peut être nécessaire pour les textes non ASCII dans certains shells (par exemple : si l'affichage est défaillant, essayez `utf8` ou un autre encodage).

`PYTHONHASHSEED`

Si la variable a la valeur « random », une valeur aléatoire sera utilisée pour le hachage des objets `str`, `bytes` et `datetime` ; il est également possible d'attribuer à la variable un entier dans la plage allant de `0` à `4 294 967 295` pour obtenir des valeurs de hachage prévisibles (à partir de Python 3.2.3 et 2.6.8).

`PYTHONFAULTHANDLER`

Si la variable est définie, Python enregistre les gestionnaires dès le démarrage pour générer un objet traceback en cas d'erreurs fatales (à partir de Python 3.3, équivalent à `-X faulthandler`).

Variables de la commande Python

Les variables d'environnement suivantes sont équivalentes à certaines options de la commande Python (voir la section *Options de la commande Python*) :

`PYTHONDEBUG`
: Si elle n'est pas vide, équivaut à l'option `-d`.

`PYTHONDONTWRITEBYTECODE`
: Si elle n'est pas vide, équivaut à l'option `-B`.

`PYTHONINSPECT`
: Si elle n'est pas vide, équivaut à l'option `-i`.

`PYTHONNOUSERSITE`
: Si elle n'est pas vide, équivaut à l'option `-s`.

`PYTHONOPTIMIZE`
: Si elle n'est pas vide, équivaut à l'option `-O`.

`PYTHONUNBUFFERED`
: Si elle n'est pas vide, équivaut à l'option `-u`.

`PYTHONVERBOSE`
: Si elle n'est pas vide, équivaut à l'option `-v`.

`PYTHONWARNINGS`
: Si elle n'est pas vide, équivaut à l'option `-W`, avec la même valeur. La variable accepte aussi une chaîne séparée par des virgules comme un équivalent des options multiples de `-W` (À partir de Python 3.2 et 2.7.).

UTILISATION DU LANCEUR WINDOWS

Uniquement sous Windows, Python 3.3 (et les versions ultérieures) installe un lanceur de scripts, qui est aussi disponible séparément pour les versions précédentes. Ce lanceur se compose des exécutables `py.exe` (console) et `pyw.exe` (interface graphique), qui peuvent être invoqués sans le paramètre `PATH` ; ces exécutables sont configurés de telle sorte qu'ils peuvent exécuter des fichiers Python en se basant sur leur extension. Cela permet aux versions de Python d'être sélectionnées de trois manières différentes : avec les directives de type Unix « `#!` » au sommet des scripts, avec les arguments de la ligne de commande, et grâce à des valeurs par défaut configurables.

Directives de fichiers du lanceur

Le lanceur reconnaît les lignes « #! » en en-tête des fichiers de scripts qui indiquent les versions de Python sous l'une des formes suivantes et dans lesquelles le caractère * est égal à : *vide* pour utiliser la version par défaut (actuellement, il s'agit de la version 2, si elle est installée, et cela revient à l'omission d'une ligne « #! ») ; un numéro de version *majeure* (par exemple, 3) pour lancer la dernière version de cette ligne de produit installée ; ou bien une spécification *complète majeure.mineure*, éventuellement suffixée par 32 pour privilégier une installation 32 bits (par exemple, 3.1–32) :

```
#!/usr/bin/env python*
#!/usr/bin/python*
#!/usr/local/bin/python*
#!python*
```

Tous les arguments de la commande Python (python.exe) peuvent être indiqués à la fin de la ligne, et Python 3.4 (et les versions suivantes) peuvent interroger la variable PATH pour rechercher les lignes « #! » qui mentionnent juste python sans numéro explicite de version.

Lignes de commandes du lanceur

Le lanceur peut également être invoqué à partir du shell avec des lignes de commandes sous la forme suivante :

```
py [pyarg] [pythonarg*] script.py [scriptarg*]
```

Plus généralement, tout ce qui peut apparaître dans une commande python après son composant *python* peut aussi apparaître après l'élément optionnel *pyarg* d'une commande py, puis est passé au Python généré tel quel. Cela inclut les paramètres de spécification du programme -m, -c et - ; voir la section *Utilisation de la ligne de commande Python*.

Le lanceur accepte les formes d'arguments suivantes pour le paramètre optionnel pyarg, qui est identique à la partie * qui mentionne un fichier à la fin d'une ligne « #! » :

```
2          Lance la dernière version 2.X installée
-3         Lance la dernière version 3.X installée
-X.Y       Lance la version spécifiée (X est égal à 2 ou 3)
-X.Y32     Lance la version 32 bits spécifiée
```

Si les deux arguments sont présents, les arguments de la ligne de commande ont la priorité sur les valeurs fournies dans les lignes « #! ». Quand elles sont installées, les lignes « #! » peuvent être appliquées dans plus de contextes (par exemple, des clics sur des icônes).

Variables d'environnement du lanceur

Le lanceur reconnaît aussi les paramètres optionnels des variables d'environnement, qui peuvent être utilisés pour personnaliser le choix de la version par défaut ou quand l'information passée est partielle (par exemple, quand le numéro de version est manquant ou que seul le numéro majeur de version est précisé, ou avec l'argument de la commande `py`) :

```
PY_PYTHON   Version à utiliser par défaut(sinon 2)
PY_PYTHON3  Version à utiliser pour une version 3 partielle
(par exemple, 3.2)
PY_PYTHON2  Version à utiliser pour une version 2 partielle
(par exemple, 2.6)
```

Ces paramètres ne sont employés que par les exécutables du lanceur, et non pas quand *`python`* est invoqué directement.

TYPES INTÉGRÉS ET OPÉRATEURS

Priorité des opérateurs

Le tableau 1 *Priorités des opérateurs en Python 3.X* liste les opérateurs Python. Les opérateurs dans les cellules du bas du tableau ont une priorité plus élevée quand ils sont utilisés dans des expressions regroupant plusieurs opérateurs sans parenthèses.

Termes atomiques et typage dynamique

Dans le tableau 1 *Priorités des opérateurs en Python 3.X*, les éléments remplaçables `X`, `Y`, `Z`, `i`, `j`, et `k` peuvent prendre dans les expressions les valeurs suivantes :

- *Nom de variable*, remplacé par la dernière valeur assignée

- *Expression littérale*, définie dans la section *Types intégrés spécifiques*
- *Expression imbriquée*, provenant d'une ligne quelconque de ce tableau, éventuellement entre parenthèses

Les *variables* Python suivent un *modèle de typage dynamique* (elles ne sont pas déclarées, mais sont créées au moment de leur assignation) ; leurs valeurs sont des références d'objets, qui peuvent être de n'importe quel type. Elles doivent être assignées avant d'apparaître dans une expression, car elles n'ont pas de valeur par défaut. Dans les noms de variables, on fait toujours la différence entre les minuscules et les majuscules (voir la section *Règles de nommage*). Les *objets* référencés par des variables sont créés automatiquement, et quand ils ne sont plus utilisés, ils sont supprimés automatiquement par le *garbage collector* de Python, qui emploie des compteurs de références dans Cpython.

Également dans le tableau 1, l'élément modifiable `attr` doit être le nom littéral (sans guillemets) d'un attribut ; `args1` est une liste d'arguments formels (voir la la définition dans la section *Instruction def*) ; `args2` est une liste d'arguments d'entrée (voir la définition dans la section *Instruction expression*) ; et le littéral ... est une expression atomique seulement valide en 3.X.

La syntaxe des compréhensions et des littéraux de structures de données (tuple, liste, dictionnaire et ensemble) indiquée de manière abstraite dans les trois dernières lignes du tableau 1 est définie dans la section *Types intégrés spécifiques*.

Notes sur l'utilisation des opérateurs

- En Python 2.X seulement, l'opérateur d'inégalité peut s'écrire `X != Y` ou `X <> Y`. En Python 3.X, la dernière formulation a été supprimée car elle est redondante.
- En Python 2.X seulement, une expression entre apostrophes d'ouverture ‘`X`‘ est équivalente à `repr(X)`, et convertit les objets à afficher en tant que chaînes de caractères. En Python 3.X, il est préférable d'utiliser les fonctions intégrées `str()` et `repr()` qui sont plus lisibles.
- En Python 3.X et 2.X, la division entière `X // Y` tronque toujours la partie décimale et retourne le résultat sous la forme d'un entier pour les entiers.