Le langage Python

Éric Jacoboni

15 octobre 2021

jacoboni@univ-tlse2.fr

Sommaire

Présentation rapide du langage

Historique

- Créé par *Guido Van Rossum* en 1989. Le nom du langage vient des Monty Python (troupe d'humoristes britanniques).
- Création de la Python Software Foundation en 2001.
- Correction de certains défauts du langage avec l'apparition de Python 3.x (au prix d'une non compatibilité avec Python 2.x).
- Quasiment tous les modules Python 2.x ont été portés sous Python 3.x et il est fortement conseillé de passer à cette version du langage pour les nouveaux développements.
- Guido Von Rossum a été employé par Google en 2005 pour travailler sur Python. Il a rejoint Dropbox depuis 2013.
- Le développement du langage est maintenant géré par la PSF, dont Guido Von Rossum est le « Benevolent Dictator For Life »(BDFL).

Caractéristiques essentielles du langage

- En 1999, Van Rossum décrivait les caractéristiques essentielles du langage :
  - Facile à apprendre et intuitif, tout en étant aussi puissant que ses principaux rivaux.
  - « Open Source » afin que tout le monde puisse participer à son évolution.
  - Aussi lisible que de l'anglais classique.
  - Adapté aux tâches quotidiennes afin d'avoir des temps de développement courts.
- Python est désormais le 1er langage de programmation le plus utilisé sur les plateformes open-source.
- Il est dans le top 10 des langages les plus demandés dans les CV américains.

Remarque

Ces classements ne doivent pas être pris « pour argent content » (Objective-C, par exemple, a longtemps été 3^{e} car c'était LE langage de développement pour les systèmes Apple... Il est maintenant 19^{e} et son remplaçant, Swift, est maintenant classé 12^{e}...).

Adoption du langage

- Python est utilisé par Google, Yahoo, la NASA, et de nombreuses sociétés de développement.
- Il sert de langage de commande dans plusieurs logiciels libres : FreeCAD, Blender, Inkscape, XBMC, etc.
- De nombreux logiciels sont écrits en Python : Hotot (client Twitter), Calibre (outil de conversion et de gestion de livres électroniques), etc.
- C'est actuellement le langage de script le plus utilisé pour l'administration système et réseau sur les distributions Linux (où il a tendance à remplacer Perl). C'est donc à la fois un langage pour les développeurs et un langage pour les administrateurs.
- Depuis 2013, Python est enseigné dans les prépas (CPGE) scientifiques françaises (et dans d'autres pays). C'est donc un langage qui sera de plus en plus présent dans les projets scientifiques.
- Un bon point de départ : `http://docs.python.org/fr/3/tutorial/`

Interpréteurs interactifs

- La distribution standard de Python fournit *idle* (écrit lui-même en Python avec la bibliothèque *tkinter*). C'est un environnement interactif permettant d'exécuter du code Python.
- La commande *python* (ou *python3*) sans paramètre, lance un interpréteur interactif en mode texte.
- L'environnement *ipython* (qui s'appelle maintenant *Jupyter*) offre un mode interactif en mode texte un peu plus riche que l'interpréteur fourni avec la distribution. Son mode *notebook* permet d'utiliser un navigateur web pour exécuter du code (intéressant pour visualiser des courbes de fonctions mathématiques ou réaliser des tutoriels). Son mode *qtconsole* est une bonne alternative à *idle*.

Outils de développement

- Parmi les environnements de programmation spécialisés, les plus utilisés sont *PyDev*, un plugin pour Eclipse (voir `pydev.org`) ou *PyCharm* (voir `jetbrains.com`). Visual Code dispose également d'un plugin Python très efficace.
- Toutes les distributions Linux disposent de paquetages pour Python et il est même souvent installé par défaut (mais attention à la version du langage)
- Mac OS fournit par défaut Python 2.7.x, mais on peut installer la version 3.x de ActivePython « community edition » (voir `www.activestate.com` et les termes de la licence) ou, mieux, utiliser un paquetage *homebrew*.
- Outre la distribution officielle du site de Python, les utilisateurs de Windows peuvent également télécharger la distribution « community » d'ActivePython ou miniconda3, qui est plus complet (voir `continuum.io/blog/anaconda-python-3`).

Interpréteurs disponibles

Outre l'interpréteur « de référence » *cpython*, il existe d'autres implémentations de Python (pour l'instant essentiellement compatibles avec la version 2.7 du langage) :

- *IronPython* (`http://ironpython.net/`) est une implémentation de Python pour l'environnement .NET (et Mono). Son développement semble stagner (dernière version en décembre 2014)
- *Jython* (`http://www.jython.org/`) est une implémentation pour la plateforme Java. Il est notamment utilisé sur le serveur d'applications WebSphere d'IBM.
- *Pypy* (`http://pypy.org/`) est un interpréteur JIT (disponible pour 2.7 et 3) permettant d'accélérer l'exécution de scripts Python. Son développement est très actif.

De quoi aurons-nous besoin ?

- D'un interpréteur Python 3.6 ou supérieur et d'un terminal de commandes...
- D'un éditeur de texte avec, de préférence, la coloration syntaxique pour Python (*Emacs*, *Sublime Text*, *Atom*, *Visual Code* ou *Notepad++*, par exemple) – Une bonne solution consiste également à utiliser *PyCharm*.
- Éventuellement, *ipython* (qui s'appelle maintenant *Jupyter*) pour disposer d'un interpréteur interactif évolué.
- Un navigateur avec un favori vers la documentation de Python 3 : `http://docs.python.org/3/index.html`

Conventions de nommage

- Le *PEP 8* (PEP signifie *Python Enhancement Proposals*) définit les conventions de nommage que devraient suivre les programmeurs Python.
- L'indentation (qui définit la structure d'un programme Python) doit être de 4 espaces (les espaces sont préférés aux tabulations et on ne peut pas mélanger les deux – configurez votre éditeur de texte).
- Les lignes ne doivent pas dépasser 79 caractères. Utiliser les parenthèses ou l'anti-slash pour couper une longue ligne en plusieurs lignes.
- Séparez les définitions de fonctions/méthodes par une ligne blanche et pour séparer les blocs de code logiques.
- L'encodage des caractères est UTF-8 (configurez votre éditeur).
- Si vous comptez publier votre code, utilisez des identificateurs anglais.
- Lisez ce PEP (`http://www.python.org/dev/peps/pep-0008/`) et efforcez-vous de le respecter (les éditeurs spécialisés vous signaleront vos écarts...).

Noms des identificateurs

- Les noms des modules doivent être courts et tout en minuscules avec, éventuellement, des blancs soulignés pour séparer les mots. Attention : les noms de modules correspondent aux noms de fichiers, donc ces noms doivent être reconnus par le système d'exploitation (Unix fait la différence entre majuscules et minuscules, pas Windows).
- Les noms de classe doivent être en *CamelCase*.
- Les noms de fonctions/méthodes et les noms de variables/paramètres doivent être en *snake_case* (c'est donc différent de Java). Les noms des méthodes et variables non publiques doivent commencer par un blanc souligné. On utilise couramment deux blancs soulignés pour les méthodes et variables privées (voir plus loin).
- Pour les méthodes, utilisez *self* comme paramètre désignant l'instance et *cls* comme paramètre désignant la classe (vous pourriez choisir n'importe quels autres noms, mais ce serait une très mauvaise idée car c'est une convention unanimement respectée).

Commentaires et docstrings

- Les commentaires sont introduits par le caractère # et se terminent à la fin de la ligne.
- Utilisez des commentaires judicieux et non redondants. Si votre code doit être publié, écrivez-les en anglais.
- Un commentaire doit précéder ce qu'il commente et être au même niveau d'indentation. Un commentaire peut également être placé sur la même ligne qu'une instruction qu'il commente. Un commentaire *explique* une portion de code
- Un docstring permet de produire la documentation de votre code (voir PEP 257).
- Utilisez les triples guillemets (""") pour entourer les docstrings.
- Un docstring décrit un module, une classe ou une fonction/méthode. Ce n'est pas un commentaire, mais une *documentation*.
- Un docstring apparait toujours comme la première ligne de ce qu'il décrit (il est donc placé « après », pas « avant »).
- Un docstring est accessible via l'attribut *__doc__* de l'objet concerné.

Types de données

Entiers et chaînes de caractères

En Python, les entiers sont du type *int* et les chaînes de caractères sont du type *str*. Les caractères sont codés en Unicode :

```
-973
21062458333711437339583605536734086463779019080109822250862195507 2
0
"Tout va bien"
'Il était une fois'
''
```

- La taille des entiers est uniquement limitée par la mémoire disponible.
- Les chaînes sont délimitées par des apostrophes ou des guillemets.
- On accède à un caractère particulier en indiquant son indice entre crochets (à partir de 0) :

```
>>> "Tout va bien"[2]     => "u" (un caractère est une chaîne d'un seul caractère)
```

Entiers et chaînes de caractères

- Les chaînes et les types numériques de Python sont *immutables* : lorsqu'une variable a été initialisée, on ne peut plus modifier sa valeur.
- On ne peut donc pas écrire *"Tout va bien"[2] = "o"*...
- Pour convertir une chaîne en entier, on utilise *int(chaine)*. Pour convertir un entier en chaîne, on utilise *str(entier)* :

```
>>> int("42")           => 42
>>> int("    12  ")     => 12
>>> str(666)            => '666'
```

Remarque : Ces conversions peuvent échouer si leur paramètre n'est pas correct. En ce cas, elles lèvent une exception (voir plus loin).

Objets et références

- En Python, les « variables » sont en réalité des *références d'objets*.
- L'opérateur $=$ ne place pas une valeur *dans* une variable mais *lie* une référence (la « variable ») à un objet en mémoire.
- Si la référence était déjà liée, elle est simplement liée à un nouvel objet (l'ancien est perdu). Si elle n'existait pas, elle est créée.
- En pratique, ceci ne pose pas de problème pour les objets immutables, mais il faut le savoir pour les objets modifiables (voir plus loin).
- Les identificateurs de variables ne doivent pas être un mot-clé et doivent commencer par une lettre ou un blanc souligné, suivi éventuellement d'un ou plusieurs caractères non espace (lettre, chiffre, blanc souligné). Il n'y a pas de limite de longueur et Python est sensible à la casse.

Objets et références

- Python utilise un *typage dynamique* : une référence liée à un type d'objet donné (un *int*, par exemple) peut être liée ensuite à un autre type d'objet (un *str*, par exemple). Ce n'est pas pour ça qu'il faut le faire : l'intérêt est surtout de ne pas devoir « déclarer » les variables, comme dans les langages à type statique (C, Java, etc.).
- Les opérations applicables à une référence dépendent du type de l'objet auquel elle est liée (on ne peut pas diviser deux chaînes, par exemple : l'exception *TypeError* serait alors déclenchée).
- La fonction *type()* permet de connaître le type d'un objet à un instant donné (en pratique, elle n'est utilisée que pour le débogage) :

```
>>> route = 66
>>> print(route, type(route))   => affiche : 66 <class 'int'>

>>> route = "Nord"
>>> print(route, type(route))   => affiche : Nord <class 'str'>
```

Collections : tuples et listes

- Python permet de créer des *tuples*, des *listes*, des *tableaux associatifs* (dictionnaires) et des *ensembles*.
- Les tuples (type *tuple*) sont immutables alors que les listes (type *list*) sont des collections modifiables. Les dictionnaires et les ensembles seront présentés plus loin.
- Les tuples sont créés par des virgules et généralement placés entre parenthèses : *("coucou", 42, "bye")* est un tuple de 3 éléments qui ne pourront plus être modifiés, *("un",)* est un tuple d'un seul élément – notez la virgule – et *()* est le tuple vide.
- Dans certains contextes, on peut omettre les parenthèses autour d'un tuple.
- Les listes sont créées par des crochets : *["coucou", 42, "bye"]* est une liste de 3 éléments qui pourra être modifiée, *[]* est la liste vide.

```
>>> truc = (42, "coucou")
>>> print(truc, type(truc))   => affiche : (42, 'coucou') <class 'tuple'>

>>> truc = [42, "coucou"]
>>> print(truc, type(truc))   => affiche : [42, 'coucou'] <class 'list'>
```

Tuples et listes

- En réalité, les tuples et les listes ne « contiennent » pas d'éléments mais des *références* (ceci a des conséquences sur les copies).
- Un tuple ou une liste étant un objet comme un autre, un tuple peut contenir un tuple ou une liste (idem pour une liste).
- La fonction *len()* permet de connaître le nombre d'éléments d'un tuple, d'une liste ou d'une chaîne (car ce sont des objets « itérables ») :

```
>>> len(("un",))                => 1
>>> len("un")                   => 2
>>> len([3, 2, "coucou", 42])   => 4
>>> len("automatique")          => 11
>>> len(3)                      => TypeError: object of type 'int' has no len()
```

Tuples et listes

Un objet *list* dispose de la méthode *append()* et redéfinit l'opérateur d'addition $+$:

Ajout en fin de liste

```
>>> x = ["coucou", 42, 34, "Toulouse", 100]
>>> x.append("Plus")        => x vaut ["coucou", 42, 34, "Toulouse", 100, "Plus"]
>>> x += ["Moins"]          => x vaut ["coucou", 42, 34, "Toulouse", 100, "Plus",
                                       "Moins"]
```

Les listes disposent de l'opérateur *[]* (les indices sont contrôlés) :

Accès/modification d'un élément d'une liste

```
>>> x[0]                     => renvoie "coucou"
>>> x[1] = "quarante-deux"   => x vaut ["coucou", "quarante-deux", 34,
                                        "Toulouse", 100, "Plus"]
```

Les listes disposent de nombreuses autres méthodes, dont *insert()* et *remove()* (voir plus loin).

Opérateurs

Affectation

- On rappelle (encore une fois...) que l'affectation en Python ne stocke pas une valeur *dans* une variable, mais *lie* un nom (une référence) à une valeur (un objet).
- Comme en C, l'affectation est un *opérateur* qui renvoie une valeur (celle qui a été affectée) : on peut donc enchaîner les affectations.
- Outre l'affectation combinée aux opérations (comme $+=$), Python permet d'effectuer des *affectations en parallèle* (qui sont en fait des affectations de tuples ou de listes).
- Grâce à l'opérateur *, l'affectation peut capturer dans une liste un ensemble de valeurs (*unpacking*).

Exemples d'affectations

```
a = b = c = 0              # L'affectation = renvoie une valeur
a, b = b, a                # Échange les valeurs de a et b
a, b, c = [1, 2]           # erreur (idem avec (1, 2) ou 1, 2)
a, b, *c = [1, 2, 3, 4]    # a = 1, b = 2, c = [3, 4]
a, *b, c = [1, 2, 3, 4]    # a = 1, b = [2, 3], c = 4
a, b, *c = 1, 2, 3, 4, 5   # idem
a, b = 1, 2, 3, 4          # erreur (idem avec (1, 2, 3, 4) ou [1, 2, 3, 4])
```

L'opérateur is

- Les variables Python étant en réalité des références, il faut différencier les comparaisons de ces références et les comparaisons des objets qu'elles désignent (leur « contenu »)...
- L'opérateur *is* renvoie *True* si deux références désignent le même objet, *False* sinon (c'est donc un peu l'équivalent du $==$ de Java).
- On utilise le plus souvent *is* pour comparer une référence avec *None*, qui désigne l'objet nul :

L'opérateur is

```
>>> a = ["exemple", 42, None]
>>> b = ["exemple", 42, None]
>>> a is b                          => False
>>> b = a
>>> a is b                          => True

>>> a = "un truc"
>>> b = None
>>> a is not None, b is None        => (True, True)
```

Opérateurs de comparaison

- On dispose des opérateurs de comparaison classiques : $<$, $<=$, $==$, $!=$, $>=$ et $>$.
- Tous ces opérateurs comparent des *valeurs* d'objets, c'est-à-dire les objets désignés par les références (le $==$ de Python est donc l'équivalent du *equals()* de Java).
- La comparaison de deux valeurs de types incohérents lève l'exception *TypeError*.

Opérateurs de comparaison

```
>>> a = "un truc"
>>> b = "un truc"
>>> a is b                  => False
>>> a == b                  => True
>>> a > "un machin"         => True   (attention avec les caractères accentués)
>>> c = 9
>>> 0 <= c <= 10            => True
>>> "trois" < 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: str() < int()
```

Opérateurs d'appartenance

- Les opérateurs *in* et *not in* permettent de tester l'appartenance d'un objet à une séquence ou une collection (chaîne, liste, tuple, dictionnaire, ensemble).
- Dans le cas d'une liste ou d'un tuple, *in* effectue une recherche *linéaire* (qui dépend donc de la longueur de la collection). Cette recherche est très rapide pour les dictionnaires et les ensembles et ne dépend pas de leur nombre d'éléments (voir plus loin).

Opérateurs in et not in

```
>>> a = (42, "coucou", 9, -33, 10)
>>> 9 in a                                  => True
>>> "bla" not in a                          => True
>>> texte = "Un chasseur sachant chasser"
>>> "s" in texte                            => True
>>> "ant" in texte                          => True
```

Opérateurs logiques

- Les trois opérateurs logiques sont *and*, *or* et *not*.
- *and* et *or* fonctionnent en *court-circuit* : ils ne renvoient pas un booléen mais la valeur de l'opérande qui a déterminé le résultat.
- Dans un contexte booléen (dans un test, par exemple), le résultat de ces opérateurs est évalué comme un booléen : *0*, *""*, *[]* et *()* sont interprétés comme *False* et le reste comme *True* (voir plus loin).
- *not* renvoie toujours un booléen.

Opérateurs logiques

```
>>> cinq = 5
>>> deux = 2
>>> zero = 0
>>> cinq and deux                => 2  # a and b = if a then b else a
>>> deux and cinq                => 5
>>> cinq and zero                => 0
>>> cinq or deux                 => 5  # a or b = if a then a else b
>>> deux or cinq                 => 2
>>> zero or cinq                 => 5
>>> zero or 0                    => 0
>>> not zero                     => True
>>> not deux                     => False
```

Opérateurs arithmétiques

- Python dispose des opérateurs +, -, *, / (vraie division), // (division entière), % (modulo) et ** (puissance).
- Il fournit également les opérateurs combinés à l'affectation (+=, -=, etc.).
- Les opérateurs de pré/post incrémentation/décrémentation (++ et --) n'existent pas en Python !
- L'opérateur / effectue *toujours* une division réelle, même si les deux opérandes sont des entiers .Le type de son résultat est toujours *float*.
- L'opérateur // renvoie le *quotient*. Le type de son résultat dépend de celui de ses opérandes.

```
>>> 4/2
2.0
>>> 4//2
2
>>> 5/2
2.5
>>> 5//2
2
>>> 5.0/2
2.5
>>> 5.0//2
2.0
```

Opérateur + pour les chaînes, les tuples et les listes

- L'opérateur $+$ est redéfini pour les chaînes, les tuples et les listes afin d'effectuer une concaténation.
- Que ce soit pour une chaîne, un tuple (immutables) ou pour une liste (modifiable), l'opérateur $+$ ne modifie pas l'objet, mais en renvoie un autre – *append()*, par contre, modifie une liste sur place.
- L'opérande droite de $+$ pour les listes doit être une liste (un *iterable*, en fait). Pour les tuples, cet opérande doit etre un tuple.

Exemples

```
>>> texte = "Hello"
>>> texte + " World"                  # renvoie "Hello World"
>>> liste = ["hello", 42, "bye"]
>>> liste + "bla"                     # TypeError : can only concatenate
                                        list (not "str") to list
>>> liste + ["bla"]                   # renvoie ['hello', 42, 'bye', 'bla']
>>> liste + [66, "coucou"]            # renvoie ['hello', 42, 'bye', 66, 'coucou']
>>> couple = (10, 42)                 # ou : couple = 10, 42
>>> couple + (3,)                     # renvoie (10, 42, 3)
>>> texte                             # 'Hello'
>>> liste                             # ["hello", 42, "bye"]
>>> couple                            # (10, 42)
```

Opérateur * pour les chaines, les tuples et les listes

L'opérateur * peut également s'appliquer à une chaine, un tuple ou une liste et renvoie une autre chaine, un autre tuple ou une autre liste :

Exemples

```
>>> texte = "bla"
>>> print(texte * 3)
blablabla
>>> print("*" * 80)
********************************************************************************
>>> liste = [1]
>>> print(liste * 10)
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
>>> (10, 12) * 3
(10, 12, 10, 12, 10, 12)
>>> liste = [None] * 100     # Donne une taille initiale à liste dès sa création
```

Structures de contrôle

L'instruction if

Syntaxe

```
if expr_booléenne1:
  instructions
[elif expr_booléenne2:
  instructions
...
else:
  instructions]
```

- Il peut y avoir zéro ou plusieurs clauses *elif*.
- La clause *else* est facultative.
- Contrairement à C/C++/Java, on ne met pas de parenthèses autour du test.
- Ne pas oublier les deux-points (`:`) après chaque clause...
- Attention à l'indentation !

L'expression if

Syntaxe

```
valeur1 if expr_booléenne else valeur2
```

- C'est un peu l'équivalent de l'opérateur ternaire du langage C...
- Cette expression renverra `valeur1` si l'expression booléenne est vraie, elle renverra `valeur2` sinon.
- La partie *else* est obligatoire.
- Il n'y a pas de deux-points après l'expression booléenne, ni après le else.
- Exemples d'utilisation :

```
...
result = 42 if test > 0 else 64
...
return n if n < 2 else n + 1
```

L'instruction while

Syntaxe

```
while expr_booléenne:
  instructions
[else:
  instructions]
```

- Une instruction *break* dans le corps de la boucle fait sortir de cette boucle.
- Une instruction *continue* fait revenir au début de la boucle.
- Généralement *break* et *continue* sont utilisées avec un *if*.
- La clause *else* est facultative : ses instructions seront exécutées après la fin de la boucle (sauf si on en sort par un *break*). Si on ne rentre pas dans la boucle, le *else* est quand même exécuté. Elle est peu utilisée...
- Ne pas oublier le deux-points...

L'instruction for

Syntaxe

```
for variable(s) in itérable:
  instructions
[else:
  instructions]
```

- Ici, *itérable* désigne tout type de données pouvant être parcouru (chaînes, listes, tuples et autres types collections).
- Cet itérable peut être un littéral (`[42, "toto", -10]`, `"coucou"` ou `(10, 3, 100)`, par exemple), une variable désignant un itérable ou un appel de fonction renvoyant un itérable (comme `range()`).
- *variable* prendra successivement chaque valeur de l'itérable.
- La clause *else* facultative fonctionne comme pour le *while*.
- Ne pas oublier le deux-points...

Exemple de boucle for

```
import string

# string.ascii_lowercase renvoie la chaîne "abcdefg... xyz"
for car in string.ascii_lowercase:
  if car in "aeiouy":
    print(car, "est une voyelle")
  else:
    print(car, "est une consonne")

# Affiche tous les nombres pairs strictement inférieurs à 100
for nb in range(1, 100):
  if nb % 2 == 0:
    print(nb, end=", ")
print()

# Affiche les caractères d'une chaine avec leurs indices respectifs
# enumerate(itérable) renvoie une liste de couples (indice, élément)
texte = "hello"
for indice, car in enumerate(texte):
   print(f"{car} est à l'indice {indice}")
```

Introduction aux exceptions

```
try:
  instructions
except exception1 [as variable1]:
  instructions
...
except exceptionN [as variableN]:
  instructions
[finally:
  instructions]
```

- Si toutes les instructions de la clause *try* s'exécutent sans déclencher d'exception, les clauses *except* sont ignorées.
- Si une exception se déclenche dans la clause *try*, on exécute les instructions du premier bloc *except* qui lui correspond.
- Dans un bloc *except*, la variable (si elle a été fournie) désigne l'objet exception qui a déclenché ce bloc.
- Si une exception est déclenchée dans un bloc *except* ou si aucun *except* ne correspond à l'exception déclenchée dans le *try*, Python « remonte » jusqu'à trouver un bloc *except* qui correspond. S'il n'en trouve pas, le programme se termine avec une exception non traitée.
- Si la clause *finally* est présente, ses instructions seront toujours exécutées, qu'il y ait eu une exception ou non.

Exemple

Exemple de gestion des exceptions

```
saisie = input("Entrez un entier : ")
try:
  valeur = int(saisie)  # peut lever ValueError...
  print("Vous avez saisi", valeur)
except ValueError as erreur:
  print(erreur)
  exit(1)               # pas la peine de continuer...
print("Suite du programme")
```

Exemple d'exécution

```
Entrez un entier : 42
Vous avez saisi 42
Suite du programme

Entrez un entier : toto
invalid literal for int() with base 10: 'toto'
```

Autre exemple

Exemple de gestion des interruptions

```
while True:
  saisie = input("Entrez un entier : ")
  try:
    valeur = int(saisie)
    break
  except ValueError:                # pas besoin de la variable
    pass                            # on ne fait rien => reboucle
print("Vous avez saisi", valeur)    # ici, on est sûr que valeur est un entier
```

Exemple d'exécution

```
Entrez un entier : HJ
Entrez un entier : doiqdz
Entrez un entier : 12
Vous avez saisi 12
```

Entrées/Sorties : print

- Par défaut, la fonction *print* affiche une chaîne (ou tout ce qui peut être transformé en chaîne par *str()*) sur la sortie standard.
- Pour rediriger la sortie de *print* vers un fichier, on utilise son paramètre *file*.
- Si on lui passe plusieurs chaînes à afficher, celles-ci seront séparées par la valeur de son paramètre *sep* (qui vaut " " par défaut).
- Le paramètre *end* est une chaîne qui sera ajoutée après la dernière chaîne (il vaut `"\n"` par défaut).

Exemples

```
>>> val = 42
>>> print(val)
42
>>> import sys
>>> print("Erreur : saisie incorrecte", file=sys.stderr)
Erreur : saisie incorrecte
>>> log = open("/tmp/monlog.log", "a")
>>> print("Erreur : saisie incorrecte", file=log)
>>> log.close()
```

Entrées/Sorties : input

- La fonction *input* affiche son message sur la sortie standard, lit sur l'entrée standard et renvoie la chaîne saisie.
- Les fonctions *int()* et *float()* permettent de convertir cette saisie en nombre.

Exemples

```
>>> nom = input("Entrez votre nom : ")
Entrez votre nom : Jacoboni
>>> age = int(input("Entrez votre age : "))
Entrez votre age : 20
>>> print(f"Bonjour {nom}, vous avez {age} ans et vous êtes un menteur")
Bonjour Jacoboni, vous avez 20 ans et vous êtes un menteur
```

Définitions et appels de fonctions

- Une définition de fonction est introduite par le mot-clé *def*.
- Comme en C, les parenthèses autour de la liste des paramètres sont obligatoires (même s'il n'y a pas de paramètre).
- Les paramètres peuvent avoir des valeurs par défaut.
- Lors de l'appel de la fonction, les paramètres peuvent être passés par position ou par nom.
- Un paramètre spécial (introduit par *) peut capturer dans un tuple tous les paramètres positionnels passés à l'appel.
- Un autre par amètre spécial (introduit par **) peut capturer dans un dictionnaire tous les paramètres nommés passés à l'appel.
- Une fonction renvoie sa valeur grâce à l'instruction *return*.
- S'il n'y a pas de *return*, la fonction renvoie la valeur *None*.

Exemples

```
def fonction1(x, y, z):
  val = x + 2*y + z**2
  return val if val > 0 else 0

v1, v2 = 3, 4
fonction1(v1, v2, 2)                    # Renvoie 15
fonction1(v1, z = v2, y = 2)            # Utilisation de params nommés : renvoie 23

def fonction2(x, y = 1, z = 1):         # Utilisation de params par défaut
  return x + 2*y + z**2

fonction2(3, z = 4)                     # Renvoie 21

def fonction3(x, y = 1, z = 1, *tup):
  print(x, y, z, tup)

fonction3(2)                            # Affiche 2 1 1 ()
fonction3(1, 2, 3, 4, 5, 6)             # Affiche 1 2 3 (4, 5, 6)

def fonction4(x, y = 1, z = 1, **dict):
  print(x, y, z, dict)

fonction4(1, 2, m = 5, n = 9, z= 3)     # Affiche 1 2 3 {'n': 9, 'm': 5}
```

Variables locales et globales

- Les paramètres et les variables définies dans le corps d'une fonction sont *locaux* à l'appel de la fonction.
- En Python, le code d'une fonction ne peut pas, par défaut, modifier les variables qui sont définies à l'extérieur. Il faut les indiquer explicitement par une déclaration *global*.
- L'accès en lecture est, par contre, toujours possible (mais il est conseillé de toujours indiquer par *global* les variables globales...).
- Il existe également une déclaration *nonlocal* que nous ne présenterons pas ici...

Exemple à ne pas suivre...

```
variable_globale = 42
def mauvaise_fonction(x):
   global variable_globale
   variable_globale += x
```

Affectations de fonctions

En Python, les fonctions sont des objets comme les autres, elles peuvent donc être affectées à des variables, placées dans des listes ou des dictionnaires, passées en paramètres à d'autres fonctions, etc.

```
def fahr_to_cels(fahr):
  """Renvoie l'équivalent Celsius d'un degré Fahrenheit"""
  return (fahr - 32) / 1.8

def cels_to_fahr(cels):
   """Renvoie l'équivalent Fahrenheit d'un degré Celsius"""
   return (cels * 1.8) + 32

cels_to_fahr(0)          # 0C = 32F (point de fusion de la glace)
fahr_to_cels(100)        # 100F = 37.8C (temp approx du sang)

temperature = fahr_to_cels      # Affectation de la fonction
temperature(100)                # 37.8 (conversion en Celsius)
temperature = cels_to_fahr
temperature(100)                # 212 (conversion en Fahrenheit)

temps = {'F2C': fahr_to_cels, 'C2F': cels_to_fahr}
temps['F2C'](100)               # 37.8 (conversion en Celsius)
temps['C2F'](100)               # 212 (conversion en Fahrenheit)

def convert(temp, fonction):
  return fonction(temp)

convert(100, fahr_to_cels)      # 37.8 (conversion en Celsius)
convert(100, cels_to_fahr)      # 212 (conversion en Fahrenheit)
```

Lambda-Expressions

- Une *lambda-expression* est une fonction anonyme qui peut être utilisée partout où une fonction est attendue.
- En Python, une lambda-expression est introduite par le mot-clé *lambda*. Elle ne peut contenir qu'une seule expression.
- Une lambda-expression Python renvoie toujours la valeur de son expression (pas d'instruction *return*).
- Sa syntaxe est de la forme *lambda params : expression*. (voir plus loin)

Collections

Les listes

- Une liste Python ressemble à un tableau des autres langages. C'est une *collection ordonnée* d'objets de tous types. On la note entre crochets en séparant ses éléments par des virgules. Elle est du type *list*.
- Les listes Python sont des structures de données *modifiables* (on peut modifier leur contenu après leur création).
- Les éléments d'une même liste peuvent être de types différents.
- Les éléments sont accessibles via leurs *indices* notés entre crochets. Le premier indice est 0.
- Les indices négatifs permettent de parcourir une liste de droite à gauche (l'indice -1 est l'indice du dernier élément, -2 celui de l'avant-dernier, etc.).

Exemples

```
liste = [2, 10, "bla", 3.14]
type(liste)                  # <class 'list'>
len(liste)                   # renvoie 4
liste[4]                     # IndexError: list index out of range !
liste[3]                     # renvoie 3.14   (comme liste[-1])
liste[1]                     # renvoie 10     (comme liste[-3])
```

Les tranches

- Une tranche de liste est une portion de liste délimitée par deux indices. Une liste étant modifiable, une tranche de liste est également modifiable.
- La tranche `une_liste[deb:fin]` désigne la tranche de *une_liste* comprise entre les indices *deb* compris et *fin non compris*.
- *deb* ou *fin* (ou les deux) peuvent être omis. En ce cas, les valeurs par défaut seront, respectivement *0* et *len(une_liste)*. Donc `une_liste[:]` est la tranche contenant tous les élements de la liste.
- On peut également indiquer un pas de progression (qui est de 1 par défaut) : `une_liste[deb:fin:pas]`.

Exemples

```
liste = ["un", "deux", "trois", "quatre"]
liste[1:-1]                                  # renvoie ['deux', 'trois']
liste[:3]                                    # renvoie ['un", 'deux', 'trois']
liste[2:]                                    # renvoie ['trois', 'quatre']
liste[-2:-1]                                 # renvoie ['trois']
liste[-1:2]                                  # renvoie [] (parce que -1 est "après" 2)
liste[-1:2:-1]                               # renvoie ['quatre']
l1 = liste                                   # l1 désigne la même liste
l2 = liste[:]                                # l2 contient les mêmes éléments que liste
```

Modification d'une liste

- Le contenu d'une liste peut être modifié directement au moyen des indices et des tranches (voir exemples).
- La méthode *l1.append(elt)* ajoute *elt* à la fin de *l1*.
- La méthode *l1.extend(l2)* ajoute les éléments de *l2* à la fin de *l1*.
- La méthode *l1.insert(indice, elt)* ajoute *elt* avant *indice*.
- La méthode *l1.remove(elt)* supprime la première occurrence de *elt* dans *l1*.
- La méthode *l1.pop([indice])* renvoie et supprime l'élément à l'indice indiqué (par défaut, indice = -1).
- La méthode *l1.clear()* supprime tous les éléments de la liste.
- L'instruction *del* permet de supprimer des éléments ou des tranches.
- La méthode *l1.reverse()* renverse *l1*.
- La méthode *l1.sort(key=None, reverse=False)* trie *l1*. Le paramètre *key* peut être une fonction (ou une lambda) renvoyant le critère de tri.

Modification d'une liste

Exemples

```
liste = list(range(3, 7))       # ou [*range(3, 7)] (Python 3.5)   [3, 4, 5, 6]
liste[1] = 'hello'              # [3, 'hello', 5, 6]
liste[1:3] = 'bla'              # [3, 'b', 'l', 'a', 6]
liste[1:3] = ['bla']            # [3, 'bla', 'a', 6]
liste[1:3] = 42                 # TypeError: can only assign an iterable
liste.append([3, 7])            # [3, 'bla', 'a', 6, [3, 7]]
liste.extend([30, 70])          # [3, 'bla', 'a', 6, [3, 7], 30, 70]
liste.append(42)                # [3, 'bla', 'a', 6, [3, 7], 30, 70, 42]
liste.insert(4, 'truc')         # [3, 'bla', 'a', 6, 'truc', [3, 7], 30, 70, 42]
liste.insert(0, 1000)           # [1000, 3, 'bla', 'a', 6, 'truc', [3, 7], 30, 70, 42]
liste.remove(3)                 # [1000, 'bla', 'a', 6, 'truc', [3, 7], 30, 70, 42]
liste.pop()                     # renvoie 42 et liste = [1000, 'bla', 'a', 6, 'truc', [3, 7], 30, 70]
liste.reverse()                 # [70, 30, [3, 7], 'truc', 6, 'a', 'bla', 1000]
liste.sort()                    # impossible : les éléments ne sont pas comparables entre eux
del liste[2:3]                  # [70, 30, 'truc', 6, 'a', 'bla', 1000]
liste[2:6] = []                 # [70, 30, 1000]
liste.sort()                    # [30, 70, 1000]
liste.sort(reverse=True)        # [1000, 70, 30]
liste.clear()                   # []
```

Opérations non destructrices sur les listes

- La fonction *sorted(liste, key=None, reverse=False)* renvoie une copie de *liste* triée (en fait, *sorted* fonctionne avec tous les objets Python itérables). Les éléments de *liste* doivent être comparables entre eux.
- Les opérateurs *elt in liste* et *elt not in liste* permettent de tester l'appartenance d'un élément à une liste.
- L'opérateur *l1 + l2* renvoie la concaténation de *l1* et *l2*.
- L'opérateur *liste * nbre* permet d'initialiser une liste et de lui fixer une taille initiale (ce qui évitera les réallocations futures).
- Les fonctions *min(liste)* et *max(liste)* renvoient respectivement le plus petit et le plus grand élément de la liste (ses éléments doivent être comparables entre eux).
- La méthode *liste.index(elt)* renvoie l'indice de la première occurrence de *elt* dans *liste* (ou une exception si l'élément ne s'y trouve pas).
- La méthode *liste.count(elt)* renvoie le nombre d'occurrences de *elt* dans *liste*.

Listes en intension

- Une liste en intension est décrite par les propriétés que doivent satisfaire ses éléments (les anglais les appellent « comprehension lists »).
- La syntaxe d'une liste en intension est de la forme (la partie *if* est facultative) :

```
liste = [expression for variable in iterable if condition]
```

- Exemples :

```
nbres = [1, 2, 3, 4]
carres  = [ x * x for x in nbres ]              # [1, 4, 9, 16]
carres2 = [ x * x for x in nbres if x > 2 ]     # [9, 16]
bissextiles = [annee for annee in range(1960, 2010)
               if (annee % 4 == 0 and annee % 100 != 0) or (annee % 400 == 0)]
codes = [s + z + c for s in "MF" for z in "SLMX" for c in "BGW"
         if not (s == "F" and z == "X")]
```

Tuples

- Un tuple est une sorte de liste *non modifiable* : on ne peut pas lui ajouter/ôter d'éléments après sa création et on ne peut pas non plus les modifier.
- Un tuple est noté entre parenthèses.
- L'accès (en lecture seule...) à ses éléments (ou à une tranche du tuple) utilise les crochets, comme les listes.
- La fonction *list(un_tuple)* renvoie une liste à partir d'un tuple, tandis que la fonction *tuple(une_liste)* renvoie un tuple à partir d'une liste.

Exemples

```
x = ('a', 'b', 'c')
type(x)                         # <class 'tuple'>
x[2]                            # 'c'
x[1:]                           # ('b', 'c')
len(x)                          # 3
min(x)                          # 'a'
5 in x                          # False
'b' in x                        # True
x[2] = 'd'                      # TypeError: 'tuple' object does not support item assignment
x + x                           # ('a', 'b', 'c', 'a', 'b', 'c')
x * 2                           # ('a', 'b', 'c', 'a', 'b', 'c')
x, y = 3, 4                     # identique à (x, y) = (3, 4)
(x + y)                         # 7... ce n'est PAS un tuple
(x + y,)                        # (7,) c'est un tuple à UN élément
```

Chaînes de caractères

- Les chaînes (le type *str*) peuvent être considérées comme des listes de caractères Unicode *non modifiables*.
- Comme pour les listes, on peut donc utiliser des indices et des tranches (mais uniquement pour lire, pas pour modifier).
- La fonction *len* permet de connaître la longueur d'une chaîne.
- Les méthodes qui semblent modifier une chaîne (*upper*, par exemple) ne modifient pas la chaîne mais renvoient une valeur modifiée de celle-ci.
- Le module *string* fournit des constantes utiles : *whitespace*, *digits*, *ascii_letters*, etc.

Chaînes de caractères

Exemples

```
str1, str2 = "bonjour", "salut"
x = str1 + str2                     # x = "bonjoursalut"
x = '*' * 10                        # x = "**********"
x.upper()                           # renvoie 'BONJOUR'
"BLA".lower()                       # renvoie 'bla'
"BLA bli".title()                   # renvoie 'Bla Bli'
str1.find('nj')                     # renvoie 2  (idem str1.index)
str1.find('ob')                     # renvoie -1
str1.rfind('o')                     # renvoie 4 (idem str1.rindex)
str1.index('ob')                    # ValueError: substring not found
str1.count('o')                     # 2
str1.startswith('bo')               # True
str1.replace('o', '*')              # renvoie 'b*nj*ur'
"  bla  ".strip()                   # renvoie 'bla'
"  bla  ".rstrip()                  # renvoie '  bla'
"  bla  ".lstrip()                  # renvoie 'bla  '
type(str1)                          # <class 'str'>
x = str1.encode("utf_8")            # convertit str1 en objet bytes (suite d'octets)
type(x)                             # <class 'bytes'>
import string
string.ascii_letters                # 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
str1.translate(str1.maketrans('bj', '*+'))  # renvoie '*on+our'
```

Formatage des chaînes

- Avant Python 3.6, les chaînes pouvaient être formatées avec la méthode *format* ou avec l'opérateur *%* (ce dernier ne devrait plus être utilisé dans les nouveaux programmes).

Exemples

```
"{} est la somme de {} et {}".format(10, 7, 3)      # '10 est la somme de 7 et 3'
"{0} est la somme de {2} et {1}".format(10, 7, 3)   # '10 est la somme de 3 et 7'
"Un tiers = {}".format(1/3)                         # 'Un tiers = 0.3333333333333333'
"Un tiers = {:1.2}".format(1/3)                     # 'Un tiers = 0.33'
"%d est la somme de %d et %d" % (10, 7, 3)          # '10 est la somme de 7 et 3'
"Un tiers = %f" % ((1/3))                           # 'Un tiers = 0.333333'
"Un tiers = %1.2f" % ((1/3))                        # 'Un tiers = 0.33'
```

- Python 3.6 a ajouté les *chaînes formatées* qui permettent d'*interpoler* des expressions :

```
f"La variable val contient la valeur {val}"
f"Il y {nb} élève{'s' if nb > 1 else ''}"
f"Avec 2 chiffres après la virgule : {val:.2}"       # 0.33 si val vaut 1/3
```

Dictionnaires

- Un dictionnaire est un *tableau associatif* : chaque élément de ce tableau est accessible via sa *clé*.
- Une clé de dictionnaire peut être de n'importe quel type *hachable* et *non modifiable*, ce qui est notamment le cas des nombres, des chaînes et des tuples de valeurs hachables.
- Chaque clé du dictionnaire est *unique*.
- Les valeurs stockées dans le dictionnaire peuvent être de n'importe quel type. Contrairement aux valeurs des listes (qui ont des indices numériques séquentiels), elles ne sont pas ordonnées.
- Un dictionnaire est du type *dict*. Le dictionnaire vide est noté $\{\}$.
- L'accès (en lecture ou en écriture) à une valeur du dictionnaire utilise la notation entre crochets, comme les listes (sauf que l'on utilise une clé et non un indice).
- Alors que l'écriture à un indice inexistant d'une liste provoque une erreur, l'accès en écriture à une clé inexistante d'un dictionnaire crée une nouvelle entrée dans celui-ci.

Exemples d'utilisation

Exemples

```
en_to_fr = {}                           # création d'un dico vide
en_to_fr['red'] = 'rouge'               # nouvelles associations clé -> valeur
en_to_fr['blue'] = 'bleu'
en_to_fr['green'] = 'vert'

print("red is", en_to_fr['red'])        # Affiche 'red is rouge'

fr_to_en = { 'rouge': 'red', 'vert': 'green', 'bleu': 'blue'}
len(fr_to_en)                           # 3

list(en_to_fr)                          # ['blue', 'red', 'green']
list(en_to_fr.keys())                   # idem
list(en_to_fr.values())                 # ['bleu', 'rouge', 'vert']
list(en_to_fr.items())                  # [('blue', 'bleu'), ('red', 'rouge'), ('green', 'vert')]

for color in en_to_fr.keys():
  print(en_to_fr[color], end=', ')      # Affiche 'rouge, bleu, vert'

for color in en_to_fr:
  print(en_to_fr[color], end=', ')              # idem

for (color, couleur) in en_to_fr.items():
  print(f"{color} en français se dit {couleur}")
```

Remarques

- Les méthodes *keys*, *values* et *items* ne renvoient pas des listes, mais des *vues* dynamiques. Si l'on veut obtenir des listes, il faut donc les convertir avec *list*.
- Ces vues peuvent être parcourues comme n'importe quelle séquence, avec des *for*, des *in*, etc.
- Ces vues ne sont pas ordonnées (mais on peut les trier...).
- Si un accès en écriture à une clé inexistante crée une nouvelle entrée, un accès en lecture à une clé inexistante provoque l'exception *KeyError* (donc toujours utiliser *in* pour tester avant la présence d'une clé, ou préférer la méthode *get*).
- La fonction *del* permet de supprimer une entrée de dictionnaire.
- Comme pour les listes, on peut construire un *dictionnaire en intension*.

Exemples

```
en_to_fr['purple']                        # KeyError : 'purple'

if 'purple' in en_to_fr:
  print(en_to_fr['purple'])               # N'affichera donc rien...

print(en_to_fr.get('purple', 'inconnu'))  # Affichera 'inconnu'

for color in sorted(en_to_fr):            # Tri sur les clés
   print(en_to_fr[color], end=', ')       # Affichera 'bleu, vert, rouge'

del(en_to_fr['blue'])                     # Supprime une entrée

liste = [1, 2, 3, 4]
dico_carres = { cle: cle**2 for cle in liste }            # {1: 1, 2: 4, 3: 9, 4: 16}
dico_cubes = { cle: cle**3 for cle in liste if cle > 2}   # {3: 27, 4: 64}
```

Cas d'utilisation typiques

- *Compter les mots d'une phrase :* chaque nouveau mot devient une clé (avec une valeur de 1). Cette valeur est incrémentée à chaque nouvelle occurrence.

```
import re                   # Pour les expressions régulières

phrase = 'To be or not to be, that is the question'
occurrences = {}
for mot in re.split('\W+', phrase):
  mot = mot.lower()
  occurrences[mot] = occurrences.get(mot, 0) + 1

# Affichage du résultat
for mot in sorted(occurrences):
  print(f"Le mot {mot} apparaît {occurrences[mot]} fois")
```

Cas d'utilisation typiques

- *Utilisation comme cache :* on met dans un dictionnaire des résultats déjà calculés afin de ne pas devoir les refaire ensuite.

```
def fibo_cache(n):
    cache = {0:0, 1:1}
    def fibo_aux(n):
        if n not in cache:
            cache[n] = fibo_aux(n-2) + fibo_aux(n-1)
        return cache[n]
    return fibo_aux(n)

def fibo(n):
    return n if n <= 1 else fibo(n - 2) + fibo(n - 1)
```

- Comparaison des temps d'exécution de *fibo(35)* et *fibo_cache(35)* :

```
% python3 -m perf timeit -s 'import fibo' 'fibo.fibo(35)'
.....................
Mean +- std dev: 4.92 sec +- 0.42 sec

% python3 -m perf timeit -s 'import fibo' 'fibo.fibo_cache(35)'
.....................
Mean +- std dev: 16.8 us +- 0.8 us
```

Ensembles

- Un ensemble est une collection de données *non ordonnées* et *non dupliquées*. Comme les clés d'un dictionnaire, les valeurs d'un ensemble doivent être *hachables* et *immutables* (cas des nombres, des chaînes et des tuples, mais pas des listes, des dictionnaires et des ensembles eux-mêmes).
- Outre l'ajout d'élément (et leur suppression), les opérations sur les ensembles sont les tests d'appartenance et d'inclusion, l'union, l'intersection et la différence.
- En Python, les ensembles sont implémentés par la classe *set*, l'ajout d'un élément par la méthode *add*, sa suppression par la méthode *remove*, le test d'appartenance par les opérateurs *in* ou *not in*. Les opérations d'union, d'intersection et de différence symétrique sont, respectivement, assurées par les opérateurs `|`, `&` et `^` (ou par les méthodes *union*, *intersection* et *symmetric_difference*). La différence ensembliste est implémentée par l'opérateur `-` ou la méthode *difference* (voir la doc pour les autres opérations...).

Ensembles et test d'appartenance

- Comme pour les autres collections, la fonction *len* renvoie le nombre d'éléments (sa « cardinalité ») et la boucle *for* permet de parcourir ses éléments.
- L'ensemble vide se note *set()*.
- Comme pour les listes et les dictionnaires, on peut créer des *ensembles en intension*.
- Les ensembles étant implémentés par des hachages, ils sont particulièrement adaptés aux tests d'appartenance :

```
% python3 -m timeit -s 'li = list(range(100))' '"x" in li'
100000 loops, best of 3: 2.17 usec per loop
% python3 -m timeit -s 'li = set(range(100))' '"x" in li'
10000000 loops, best of 3: 0.0305 usec per loop
```

- Même le cas le plus favorable des listes est à peine meilleur que les ensembles :

```
% python3 -m timeit -s 'li = list(range(100))' '0 in li'
10000000 loops, best of 3: 0.0243 usec per loop
% python3 -m timeit -s 'li = set(range(100))' '0 in li'
10000000 loops, best of 3: 0.0319 usec per loop
```

Exemple

```
s = set([1, 3, 5, 7])
t = set([1, 2, 3, 4, 6, 8])
s.union(t)                    # set([1, 2, 3, 4, 5, 6, 7, 8])
s | t                         # idem
s & t                         # set([1, 3])
s - t                         # set([5, 7])
s ^ t                         # set([2, 4, 5, 6, 7, 8])
s.issubset(set(range(1,10))) # True
s.add(3)                      # s non modifié
s.remove(2)                   # KeyError
if 2 in s: s.remove(2)        # s non modifié
s.discard(2)                  # pas d'erreur et s non modifié

# On exploite le fait que les ensembles soient implémentés à l'aide de dict :

u = {1, 3, 4, 12}             # set([4, 3, 12, 1])
u = { e for e in range(1,20) if e % 2 == 0 } # ensemble en intension

# set appliqué à un dictionnaire renvoie l'ensemble de ses clés :
moi = {'prénom': 'Eric', 'nom': 'Jacoboni', 'age': 20}
champs = set(moi)             # set(['age', 'nom', 'prénom'])
```

Modules

Introduction

- Les modules permettent de découper un projet en plusieurs fichiers. Ils permettent surtout de réutiliser du code.
- Un module est un fichier contenant des définitions de fonctions, de classes et autres objets et éventuellement du code exécutable directement. Le nom du module correspond à son nom de fichier (qui porte généralement l'extension *.py*).
- Un module peut être écrit en Python ou en C/C++. Quel que soit le langage utilisé pour le coder, son utilisation sera ensuite la même.
- Les modules permettent également d'éviter les conflits de noms car on peut toujours préfixer le nom d'un objet par le nom du module dans lequel il est défini.
- Un module définit un *espace de noms* (via un dictionnaire).
- Pour utiliser un module dans un autre fichier Python, il faut utiliser l'instruction *import*.

Modules précompilés

- Pour optimiser le chargement des modules, Python stocke leur version précompilée dans le répertoire *__pycache__* sous la forme *module.version.pyc* (où *version* est la version de Python qui a compilé ce module).

Exemple

- Soit le fichier *geometrie.py* suivant :

```
""" geometrie : un exemple de module écrit en Python """

pi = 3.14159

def surface_cercle(rayon):
  """ surface_cercle(rayon) : renvoie la surface du cercle de rayon indiqué."""
  global pi
  return pi * rayon**2
```

- Exemples d'utilisation :

```
pi                                  # NameError: name 'pi' is not defined
surface_cercle(2)                   # NameError: name 'surface_cercle' is not defined

import geometrie
pi                                  # NameError: name 'pi' is not defined
geometrie.pi                        # 3.14159
geometrie.surface_cercle(2)         # 12.56636

geometrie.__doc__                   # ' geometrie : un exemple de module écrit en Python '
geometrie.surface_cercle.__doc__    # ' surface_cercle(rayon) : renvoie la surface du cercle de rayon indiqué.'

from geometrie import *             # Pas conseillé...
surface_cercle(2)                   # 12.56636

from geometrie import pi as mon_pi, surface_cercle as surf_cercle
# ou : import geometrie.pi as mon_pi, geometrie.surface_cercle as surf_cercle

surf_cercle(2)
mon_pi
```

Recherche des modules

- Python recherche les modules dans les répertoires énumérés dans la variable *path* du module *sys* :

```
import sys
print(sys.path)        # Liste de chaînes contenant les répertoires des modules
```

- Ces répertoires sont recherchés dans l'ordre : dès que le module est trouvé, la recherche s'arrête. Si le module n'est pas trouvé, Python produit une exception *ImportError*.
- Lorsque l'on exécute un script Python, le premier chemin de la liste *sys.path* est toujours celui du répertoire où se trouve ce script.
- Dans une session interactive (et donc avec iPython), le premier chemin est la chaîne vide, qui représente le répertoire d'où a été lancé la session.

Stockage de ses propres modules

Il y a plusieurs choix pour stocker ses propres modules :

- Les mettre dans l'un des répertoires de *sys.path*. C'est la solution apparemment la plus simple, mais il ne faut *jamais* l'utiliser sous peine de risquer d'écraser des modules prédéfinis...
- Les mettre dans le même répertoire que le programme qui les utilise. Cette solution convient dans le cas où ces modules ne sont utilisés que par ce programme.
- Les mettre dans un (ou plusieurs) répertoire(s) particulier(s) et ajouter ce(s) répertoire(s) à *sys.path* : soit en modifiant directement *sys.path* dans le programme utilisateur avant d'importer le(s) modules(s), soit en initialisant la variable shell *PYTHONPATH*, soit en créant un *paquetage* (voir le tutoriel ou le manuel de référence pour la création de paquetages).

Noms exportés et noms privés

- L'instruction *from module import ** importe tous les noms du module qui ne sont pas explicitement cachés (pratique à éviter car on ne sait plus à quel module appartient un nom).
- Pour cacher un nom, il suffit de le préfixer par un blanc souligné. Il ne sera alors jamais importé implicitement par *from module import ** mais restera accessible par les autres méthodes (voir exemple).
- La fonction *dir(module)* renvoie la liste des noms définis dans le module indiqué. On remarquera que certains noms sont spéciaux (ceux encadrés par deux blancs soulignés, comme *__doc__*).
- L'entrée *__name__* contient le nom du module (qui vaut *'__main__'* pour les scripts et les sessions interactives.
- L'entrée *__builtins__* contient tous les noms prédéfinis (noms des exceptions prédéfinies, nom spéciaux prédéfinis, noms des fonctions prédéfinies).

Noms exportés et noms privés

- Soit le module *mon_test.py* suivant :

```
"""Module de test des noms"""

def f(x): return x
def _g(x): return x

val1 = 42
_val2 = 64
```

- Exemple d'utilisation :

```
from mon_test import *

f(3)         # Ok, renvoie 3
_g(3)        # NameError : '_g' is not defined

val1         # Ok, 42
_val2        # NameError : '_val2' is not defined

import mon_test

dir(mon_test)   # ['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
                  '__package__', '__spec__', '_g', '_val2', 'f', 'val1']

mon_test.__name__  # 'mon_test'

mon_test._g(3)  # Ok, renvoie 3
mon_test._val2  # Ok, 64

from mon_test import _g
_g(3)           # Ok, renvoie 3
```

__name__ et '__main__'

- Lorsque l'on écrit un module, il peut être utile d'y ajouter un code permettant de le tester directement.
- Ce code doit s'exécuter si le module est lancé comme un script (avec *python3 monmodule.py*, par exemple). Sa variable *__name__* vaudra alors *'__main__'*.
- Il ne doit pas s'exécuter si le module est importé (avec *import* ou *from*). Sa variable *__name__* contiendra alors le nom du module.
- Il suffit donc de tester dans le module si *__name__* est égal à *'__main__'* :

```
"""Un module de test... """

... définition du module ...

if __name__ == '__main__':
  ... code de test du module qui ne s'exécutera que si ce fichier est lancé comme un script
```

- Ceci ne dispense pas de l'écriture de tests unitaires... (*Cf.* les exemples d'utilisation de *nose*).

Le module zipapp

- Python peut directement exécuter des archives ZIP à condition que cette archive contienne un fichier *__main__.py*.
- En pratique, ceci signifie que l'on peut donc livrer une application Python sous la forme d'un unique fichier qui aura l'extension *.pyz*.
- À partir de Python 3.5, la distribution contient un module *zipapp* permettant de créer une telle archive.
- La démarche consiste à :
  1. Regrouper dans une arborescence tous les fichiers de son application.
  2. Renommer le fichier principal en *__main__.py*.
  3. Créer l'archive en faisant *python3 -m zipapp nomrep* (ou *nomrep* est la racine de l'arborescence de l'application), ce qui aura pour effet de créer une archive *nomrep.pyz* (avec les versions précédentes de Python, il fallait créer l'archive « à la main »).
  4. Pour exécuter l'application, il suffit ensuite de faire *python3 nomrep.pyz*
- Pour en savoir plus, lire le PEP 0441.