Le langage Python

Éric Jacoboni

jacoboni@univ-tlse2.fr

21 janvier 2018

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 le 3^{e} langage de programmation le plus utilisé sur GitHub (derrière JavaScript et Java).
- Il est dans le top 10 des langages les plus demandés dans les CV américains.
- Il est actuellement 5^{e} au classement TIOBE (derrière Java, C, C++ et C#). Il était 8^{e} l'année dernière...
- Remarque personnelle : ces classements ne doivent pas être pris « pour argent content » (Objective-C, par exemple, était 3^{e} l'année dernière car c'était LE langage de développement pour les systèmes Apple... Il est maintenant 14^{e} et son remplaçant, Swift, a déjà gagné 3 places...).

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.
- Sublime Text (éditeur de texte) fournit également un interpréteur Python3 pour gérer son langage de commandes.
- 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 (en anglais) : `http://docs.python.org/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*, en revanche, 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`).
- 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 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.

Remarque

Jython et pypy3 sont installés sur mass-cara2. Pour les lancer, utilisez respectivement les commandes *jython* et *pypy3*.

De quoi aurons-nous besoin ?

- D'un interpréteur Python 3 et d'un terminal de commandes...
- D'un éditeur de texte avec, de préférence, la coloration syntaxique pour Python (*Emacs*, *Komodo Edit*, *Sublime Text* ou *Atom*, 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...).
- Votre code doit être aéré et agréable à lire !

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).
- Les constantes sont tout en majuscules avec, éventuellement, des blancs soulignés pour séparer les mots.

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'>

>>> truc = ()
>>> print(truc, type(truc))   => affiche : () <class 'tuple'>

>>> truc = []
>>> print(truc, type(truc))   => affiche : [] <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.

Exemples de divisions

```
>>> 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

Syntaxe

```
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