



HUGO LAROCHELLE

Accueil

PUBLICATIONS

UNIVERSITÉ

LIENS

PYTHON

Python a été développé par Guido van Rossum, un développeur néerlandais en 1989. Le nom Python n'a rien à voir avec les reptiles vient de la série télévisée britanique *Monty Python*. En 2005, Guido fut engagé par Google dans le but de maintenir et d'améliorer Python.

Ce tutoriel Python, rédigé par [Marc-Alexandre Côté](#) et Hugo Larochelle, donne un aperçu des aspects de base les plus importants de Python. Il correspond essentiellement à une traduction de certaines sections des tutoriels <http://docs.python.org/tutorial/> et <http://www.scipy.org/Tentative NumPy Tutorial>, avec ajouts mineurs.

TABLE DES MATIÈRES

- ▶ [CARACTÉRISTIQUES GÉNÉRALES](#)
- ▶ [SYNTAXE](#)
- ▶ [INTERPRÉTEUR](#)
- ▶ [TYPES DE BASE](#)
- ▶ [STRUCTURES DE CONTRÔLE](#)
- ▶ [DÉFINITION DES FONCTIONS](#)
- ▶ [STRUCTURES DE DONNÉES](#)
- ▶ [MODULES](#)
- ▶ [LECTURE ET ÉCRITURE DE FICHIERS](#)
- ▶ [CLASSES](#)
- ▶ [NUMPY](#)
- ▶ [DÉBOGUEUR](#)



CARACTÉRISTIQUES GÉNÉRALES

- Langage interprété
- Machine virtuelle
- Portable (Unix, Windows, Mac)
- Peut être utilisé comme *Scripting language*
- Typage dynamique (JavaScript, PHP, Ruby, ...)
- Supporte plusieurs paradigmes de programmation:
 - Orientée-objet (Java, C++, ...)
 - Impérative/Procédurale (FORTRAN, Pascal, C, ...)
 - Fonctionnelle (Haskell, Scheme, ...)

SYNTAXE

Python a été conçu pour être un langage lisible. Il vise à être visuellement épuré. Il utilise des mots anglais fréquemment là où d'autres langages utilisent de la ponctuation, et possède également moins de constructions syntaxiques que de nombreux langages structurés tels que C, Perl, ou Pascal. Les commentaires sont indiqués par le caractère '#.

Les blocs sont identifiés par l'**indentation**, au lieu d'accolades comme en C ou C++ ou de begin ... end comme en Pascal. Une augmentation de l'indentation marque le début d'un bloc, et une réduction de l'indentation marque la fin du bloc courant.

INTERPRÉTEUR

Python peut être utilisé en mode interpréteur (très pratique pour tester des lignes de code rapidement). Pour démarrer l'interpréteur Python, il suffit de taper dans une console Unix (ou DOS) la commande

```
python
```

Pour quitter, il faut taper un caractère de fin de fichier (soit `Control-D` sous Unix, `Control-Z` sous Windows) ou bien utiliser la commande `quit()`.

Il existe également une version plus riche de l'interpréteur, IPython, qui permet entre autre de faire de la complétion d'expression (en appuyant sur la touche de tabulation ou *tab*). Pour l'invoquer, suffit de taper la commande

```
ipython
```

Pour plus d'information sur IPython, visiter <http://ipython.org/>.

PASSAGE D'ARGUMENTS

La commande `python` peut également lancer l'exécution de code dans des fichiers ou scripts. Lorsqu'ils sont spécifiés à l'interpréteur, le nom du script et les arguments supplémentaires sont transformés en une liste de chaînes et affectés à la variable `argv` du module `sys`. Vous pouvez accéder à cette liste en exécutant `import sys`. La longueur de la liste est au moins un. Lorsqu'aucun script ou arguments ne sont donnés, `sys.argv[0]` est une chaîne vide.

TYPES DE BASE

NOMBRES

L'interpréteur peut servir de calculatrice: vous pouvez y taper une expression et il écrira la résultat. La syntaxe des expressions est simple: les opérateurs `+`, `-`, `*` et `/` fonctionnent exactement comme dans la plupart des autres langages (ex. Pascal ou C). S'ajoute également l'opérateur `**` pour les puissances. Les parenthèses peuvent être utilisées pour grouper les expressions. Par exemple :

```
>>> 2+2
4
>>> # Ceci est un commentaire
...
>>> 2+2 # et un commentaire sur une même ligne de code
4
>>> (50-5*6)/4
5
>>> # Division entière retourne la valeur tronquée (floor) :
...
>>> 7/3
2
>>> 7/-3
-3
>>> 2**4
16
```

Le symbole égal (`=`) est utilisé pour assigner une valeur à une variable. L'affectation ne cause aucun affichage de résultat avant la prochaine invite de commande :

```
>>> largeur = 20
>>> hauteur = 5*9
>>> largeur * hauteur
900
```

Une valeur peut être assignée simultanément à plusieurs variables :

```
>>> x = y = z = 0 # Zéro x, y et z
>>> x
0
>>> y
0
>>> z
0
```

Les variables doivent être "définies" (c.-à-d. une valeur leur a été assignée) avant de pouvoir s'en servir, sinon cela cause une erreur:

```
>>> # try to access an undefined variable
...
n
```

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

Python supporte entièrement les nombres à points flottants; les opérateurs convertissent les entiers en réels au besoin:

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

Dans la console interactive, la dernière expression calculée est assignée à la variable `_`. Ce qui veut dire que lorsque vous utilisez Python comme calculatrice, il peut être parfois utile, par exemple:

```
>>> taxe = 12.5 / 100
>>> prix = 100.50
>>> prix * taxe
12.5625
>>> prix +
113.0625
>>> round(_, 2)
113.06
```

Cette variable devrait toujours être traité comme une variable en lecture seule. N'assignez jamais explicitement une valeur à cette variable.

STRING

En plus des nombres, Python peut aussi manipuler les chaînes de caractères, qui peuvent être exprimées de plusieurs façons. Elles peuvent être entourés par des apostrophes ou des guillemets:

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

L'interpréteur affiche le résultat des opérations de chaîne de la même manière qu'ils ont été saisies. La chaîne est entre guillemets si la chaîne contient un apostrophe et aucun guillemet, sinon c'est entre apostrophe. La fonction `print` produit un affichage plus lisible.

Les continuations de lignes peuvent être utilisés avec un *backslash* en fin de ligne qui permet d'indiquer que la prochaine ligne est la suite logique de la ligne courante:

```
hello = "Ceci est plutôt une longue chaîne contenant\n\
plusieurs lignes de texte comme on peut le faire en C.\n\
Notez que les espaces au début de la ligne sont \
prises en compte."
print hello
```

Notez que les sauts de lignes ont toujours besoin d'être spécifiés dans la chaîne par le caractère `\n` - le saut de ligne suivant le *backslash* de continuation n'est pas tenu en compte. Cet exemple affichera:

```
Ceci est plutôt une longue chaîne contenant
plusieurs lignes de texte comme on peut le faire en C.
Notez que les espaces au début de la ligne sont prises en compte.
```

Par contre, les chaînes de caractères peuvent être placées entre une paire de triple-guillemets: `"""` ou de triple-apostrophes: `'''`. Il n'est pas nécessaire d'utiliser le caractère de continuation dans ce cas. Les saut de lignes seront pris en compte.

```
print """
Usage: thingy [OPTIONS]
      -h                               Display this usage message
      -H hostname                      Hostname to connect to
"""


```

affichera:

```
Usage: thingy [OPTIONS]
      -h                                     Display this usage message
      -H hostname                            Hostname to connect
```

Les chaînes peuvent être concaténées avec l'opérateur `+`, et répétées avec `*`:

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
```

Les chaînes peuvent être indexées; comme en C, le premier caractère d'une chaîne est à l'indice 0. Un caractère est simplement une chaîne de longueur un. Comme en Icon, les sous-chaînes peuvent être obtenues par l'utilisation de la *slice notation* (tranche): deux indices séparés par un deux-points `:`.

```
>>> word[4]
'A'
>>> word[0:2]
'He'
>>> word[2:4]

'lp'
```

Les indices définissant la tranche possèdent des valeurs par défaut très pratiques; l'omission du premier indice équivaut à l'indice zéro, l'omission du deuxième équivaut à la longueur de la chaîne originale.

```
>>> word[:2] # Les deux premiers caractères
'He'
>>> word[2:] # Tous les caractères sauf les deux premiers

'lpA'
```

Contrairement aux chaînes de caractères en C, celles en Python ne peuvent pas être modifiées. L'assignation à une position donnée dans la chaîne cause une erreur:

```
>>> word[0] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support item assignment
>>> word[:1] = 'Splat'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support slice assignment
```

Par contre, créer une nouvelle chaîne en combinant les différentes tranches est simple et efficace:

```
>>> 'x' + word[1:]
'xelpA'
>>> 'Splat' + word[4]
'SplatA'
```

Les indices peuvent être négatifs, permettant de compter à partir de la droite. Par exemple:

```
>>> word[-1] # Le dernier caractère
'A'
>>> word[-2] # L'avant dernier caractère
'p'
>>> word[-2:] # Les deux derniers caractères
'pA'
>>> word[:-2] # Tous les caractères sauf les deux derniers
'Hel'
```

Notez que `-0` est équivalent à `0`, donc il ne compte pas à partir de la droite!

```
>>> word[-0] # (puisque -0 égale 0)
'H'
```

Une façon de se souvenir du fonctionnement du *slicing* est d'imaginer que les indices pointent *entre* les caractères, et la gauche du premier caractère est indexée à 0. Par exemple:

```
+---+---+---+---+---+
| H | e | l | p | A |
+---+---+---+---+---+
0   1   2   3   4   5
-5  -4  -3  -2  -1
```

La fonction intégrée (*built-in*) `len()` retourne la longueur de la chaîne:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

LISTES

En Python, il existe plusieurs structures de données. La plus versatile est la liste (*list*), qui peut être créée en écrivant une suite de valeurs séparées par des virgules, le tout entre crochet. Les items d'une liste peuvent être de type différent:

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

Comme pour les chaînes de caractères, le premier item d'une liste est à l'indice 0, et les listes supportent le *slicing*, la concaténation, etc:

```
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 3*a[:3] + ['Boo!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boo!']
```

Toute opération de *slicing* retourne une nouvelle liste contenant les éléments demandés. De ce fait, la tranche suivante est une copie superficielle (*shallow copy*) de la liste *a*, c.-à-d. que les éléments de la tranche ne sont pas des copies indépendantes des éléments de la liste *a*:

```
>>> a[:]
['spam', 'eggs', 100, 1234]
```

Contrairement aux chaînes de caractères, qui sont immuables (*immutable*), il est possible de modifier individuellement les éléments d'une liste:

```
>>> a
['spam', 'eggs', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['spam', 'eggs', 123, 1234]
```

Il est également possible d'assigner des valeurs à des tranches, ce qui peut causer un changement de taille de la liste, voir même la vider complètement:

```
>>> # Replace quelques items:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Enlève quelques items:
... a[0:2] = []
>>> a
[123, 1234]
>>> # Insère quelques items:
... a[1:1] = ['bletch', 'xyzzy']
>>> a
[123, 'bletch', 'xyzzy', 1234]
>>> # Insère (une copie de) lui-même au début de la liste
>>> a[:0] = a
>>> a
```

```
[123, 'bletch', 'xyzzy', 1234, 123, 'bletch', 'xyzzy', 1234]
>>> # Vide la liste: remplace tous les items par une liste vide
>>> a[:] = []
>>> a
[]
```

La fonction intégrée `len()` s'applique également au listes:

```
>>> a = ['a', 'b', 'c', 'd']
>>> len(a)
4
```

Il est possible d'imbriquer des listes (créer de listes qui contiennent d'autres listes), par exemple:

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
>>> p[1].append('xtra')      # See section 5.1
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

Notez que l'exemple précédent, `p[1]` et `q` réfèrent au même objet!

PETIT EXEMPLE

Bien sûr, on peut utiliser Python pour autre chose qu'une simple calculatrice de poche! Par exemple, on peut calculer les premiers termes de la suite de *Fibonacci* comme suit:

```
>>> # Suite de Fibonacci:
... # la somme des deux derniers éléments de la suite définit le prochain
... a, b = 0, 1
>>> while b < 10:
...     print b
...     a, b = b, a+b
...
1
1
2
3
5
8
```

Cet exemple introduit plusieurs fonctionnalités de Python.

- La première ligne contient un *affectation multiple*: respectivement les variables `a` et `b` obtiennent simultanément les valeurs 0 et 1. Le même principe est réutilisé à la dernière ligne montrant cette fois-ci que l'évaluation de la partie droite de l'expression est faite en premier (c.-à-d. avant l'affectation).
- La boucle `while` s'exécute tant que la condition (ici: `b < 10`) demeure vrai. En Python, comme en C, n'importe quel entier non nul est évalué à vrai; zéro est faux. La condition peut également être une chaîne de caractères ou une liste, de façon plus générale n'importe quelle séquence. Si la taille est non nulle, alors l'évaluation sera vraie, sinon la séquence est vide et donc l'évaluation sera fausse. Le test utilisé dans l'exemple est une simple comparaison. Les opérateurs standards de comparaison s'écrivent comme en C: `<` (plus petit que), `>` (plus grand que), `==` (égal à), `<=` (plus petit ou égal à), `>=` (plus grand ou égal à) and `!=` (différent de).
- Le **corps** de la boucle est **indenté**: l'indentation est la technique Pythonienne de grouper les blocs d'expressions. Notez que chaque ligne d'un bloc doit avoir la même indentation.

STRUCTURES DE CONTRÔLE

Mise à part la boucle `while`, Python possède également les structures de contrôle que l'on retrouve dans les autres langages de programmation, mais avec quelques particularités.

IF

```

>>> x = int(raw_input("Please enter an integer: "))
Veuillez entrer un entier: 42
>>> if x < 0:
...     x = 0
...     print 'Negatif changé pour zéro'
... elif x == 0:
...     print 'Zéro'
... elif x == 1:
...     print 'Un'
... else:
...     print 'Plus'
...
Plus

```

Il peut y avoir aucun ou plusieurs `elif`, et le `else` est optionnel. Le mot-clé ‘`elif`’ est la contraction de ‘`else if`’

Notez qu'il n'y a pas de `switch` ni de `case` en Python.

FOR

La boucle `for` en Python est légèrement différente de celle en C ou en Pascal. Plutôt que d'itérer sur une suite arithmétique de nombres, le `for` en Python itère sur les éléments d'une séquence (ex. liste ou une chaîne de caractères) dans l'ordre qu'ils apparaissent. Par exemple:

```

>>> # Mesure quelques chaînes:
... a = ['cat', 'window', 'defenestrate']
>>> for x in a:
...     print x, len(x)
...
cat 3
window 6
defenestrate 12

```

Il n'est pas sécuritaire de modifier une séquence (ex. une liste puisqu'elle est mutable) pendant l'itération. Il est plutôt préférable d'itérer sur une copie de la liste. Pour y arriver, on peut recourir au *slicing*:

```

>>> for x in a[:]: # crée une tranche: copie entière de la liste
...     if len(x) > 6: a.insert(0, x)
...
>>> a
['defenestrate', 'cat', 'window', 'defenestrate']

```

LA FUNCTION RANGE()

Si vous avez besoin d'itérer parmi une séquence de nombres, la fonction intégrée `range()` est très pratique. Elle génère une liste contenant une suite arithmétique:

```

>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

```

L'élément de fin n'est jamais inclus dans la liste; `range(10)` génère une liste de 10 valeurs. Il est possible de spécifier le début de la suite, ou bien de modifier l'incrément (même négatif):

```

>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]

```

Pour itérer parmi les indices possibles d'une séquence, on combine `range()` et `len()`:

```

>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print i, a[i]
...
0 Mary
1 had
2 a
3 little
4 lamb

```

BREAK ET CONTINUE

Le mot-clé `break`, comme en C, termine la boucle courante: `for` ou `while`.

Le mot-clé `continue`, se comporte également comme en C et permet de continuer avec la prochaine itération de la boucle.

DÉFINITION DES FONCTIONS

On peut créer une fonction qui écrit la suite de Fibonacci jusqu'à un nombre donné comme suite:

```
>>> def fib(n):      # écrit la suite Fibonacci jusqu'à n
...     """Affiche une suite Fibonacci jusqu'à n."""
...     a, b = 0, 1
...     while a < n:
...         print a,
...         a, b = b, a+b
...
>>> # Maintenant, on appelle la fonction qu'on vient de définir:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Le mot-clé `def` introduit la définition de la fonction. Il est suivi du nom de la fonction et de la liste des paramètres entre parenthèse. Ensuite, à la ligne suivante se trouve le corps de la fonction, qui doit être indenté.

Il est simple d'écrire une fonction qui retourne la liste des nombres de Fibonacci au lieu de les afficher:

```
>>> def fib2(n): # retourne la suite de Fibonacci jusqu'à n
...     """Retourne une liste des nombres de Fibonacci jusqu'à n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)      # appel de la fonction
>>> f100                  # affichage du résultat
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Dans cet exemple, le mot-clé `return` permet à la fonction de retourner une valeur. Si la fonction ne retourne pas explicitement de valeur (par défaut), la fonction retournera `None` (peut être vu comme une sorte de pointeur nul).

ARGUMENTS PAR DÉFAUT

Ceci permet de créer une fonction qui peut être appelée avec moins d'arguments. Par exemple:

```
def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
    while True:
        ok = raw_input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise IOError('refusenik user')
    print complaint
```

La fonction peut être appelée de diverses façons:

- en spécifiant seulement l'argument obligatoire: `ask_ok('Do you really want to quit?')`
- en spécifiant un des arguments optionnels: `ask_ok('OK to overwrite the file?', 2)`
- ou bien en spécifiant tous les arguments: `ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`

Cet exemple introduit le mot-clé `in`. Il test si une certaine valeur est contenue ou pas dans une séquence.

ARGUMENTS PAR MOT-CLÉS

Les fonctions peuvent également être appelées en utilisant des *keyword arguments* de la forme `kwarg=value`. Par exemple, la fonction suivante:

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print "-- This parrot wouldn't", action,
    print "if you put", voltage, "volts through it."
```

```
print "-- Lovely plumage, the", type
print "-- It's", state, "!"
```

possède un argument obligatoire (`voltage`) et trois arguments optionnels (`state`, `action`, et `type`). Cette fonction peut être appelée de ces différentes façons:

```
parrot(1000)                                # 1 argument positionnel
parrot(voltage=1000)                          # 1 argument mot-clé
parrot(voltage=1000000, action='VOOOOOM')      # 2 arguments mot-clé
parrot(action='VOOOOOM', voltage=1000000)       # 2 arguments positionnels
parrot('a million', 'bereft of life', 'jump')   # 3 arguments positionnels
parrot('a thousand', state='pushing up the daisies') # 1 argument positionnel, 1 argument mot-clé
```

mais les appels suivants sont invalides:

```
parrot()                                     # nécessite un argument manquant
parrot(voltage=5.0, 'dead')                   # aucun argument positionnel à la suite d'un argument mot-clé
parrot(110, voltage=220)                      # affectation redondante pour un même argument
parrot(actor='John Cleese')                   # argument mot-clé inconnu
```

Lors de l'appel d'une fonction les arguments mot-clé doivent se trouver à la suite des arguments positionnels. Tous les arguments mot-clé doivent correspondre à un argument accepté par la fonction et leur ordre importe peu. Aucun argument ne peut recevoir plus d'une valeur.

STRUCTURES DE DONNÉES

LIST

Un exemple qui montre l'utilisation de la plupart des méthodes de `list`:

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.25), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
```

Outils de programmation fonctionnelle

Trois fonctions intégrées sont très utiles lorsque utilisées avec des listes: `filter()`, `map()`, et `reduce()`.

`filter(function, sequence)` retourne une séquence composée des items de `sequence` pour lesquels `function(item)` est vraie. Si `sequence` est une chaîne de caractères ou un `tuple`, le résultat sera du même type; sinon, ce sera toujours une `liste`. Par exemple, pour créer une séquence de nombres divisibles ni par 2, ni par 3:

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
...
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```

`map(function, sequence)` appelle `function(item)` pour chaque item de `sequence` et retourne une liste des valeurs de retour. Par exemple, pour calculer le cube de quelques nombres:

```
>>> def cube(x): return x*x*x
...
>>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

Plus d'une séquence peut être spécifiée; la fonction doit alors avoir autant d'arguments que de séquences. Par exemple:

```
>>> seq = range(8)
>>> def add(x, y): return x+y
...
>>> map(add, seq, seq)
[0, 2, 4, 6, 8, 10, 12, 14]
```

`reduce(function, sequence)` retourne une seule valeur créée en appelant *function* pour les deux premiers items de *sequence*. Ensuite *function* est appelée en lui passant le résultat et le prochain item de la séquence, et ainsi de suite. Par exemple, pour calculer la somme des 10 premiers nombres:

```
>>> def add(x,y): return x+y
...
>>> reduce(add, range(1, 11))
55
```

S'il n'y a qu'un seul item dans *sequence*, sa valeur est retournée; si *sequence* est vide, une exception est lancée.

Un troisième argument peut être spécifié pour indiquer la valeur de départ. Dans ce cas, cette valeur de départ est retournée si *sequence* est vide, et la fonction est d'abord appliquée sur la valeur de départ et au premier item de *sequence*, ensuite sur le résultat et le prochain item, et ainsi de suite. Par exemple,

```
>>> def sum(seq):
...     def add(x,y): return x+y
...     return reduce(add, seq, 0)
...
>>> sum(range(1, 11))
55
>>> sum([])
0
```

LISTE EN COMPRÉHENSION (*LIST COMPREHENSIONS*)

La technique liste en compréhension apporte une façon concise de créer des listes. Parmi les applications communes se trouve la confection de nouvelles listes pour lesquelles chaque élément est le résultat d'opérations effectuées sur chaque membre d'une autre séquence.

Par exemple, supposons qu'on désire créer une liste de carrés de nombre, telle:

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

On peut obtenir le même résultat avec:

```
squares = [x**2 for x in range(10)]
```

Les listes en compréhension peuvent contenir des expressions complexes et des fonctions imbriquées:

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

TUPLES ET SÉQUENCES

On voit que les listes et les chaînes de caractères ont plusieurs propriétés en commun, telles l'indexation et le *slicing*. Il y a aussi un autre type de séquence: le `tuple`.

Un tuple se compose d'un certain nombre de valeurs séparées par une virgule, par exemple:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Les tuples peuvent être imbriqués:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

Comme on le constate, à l'affichage, les tuples sont toujours entre parenthèses, afin que les tuples imbriqués soient reconnus correctement; par contre, ils peuvent être entrés avec ou sans parenthèses, malgré que les parenthèses sont souvent nécessaires malgré tout (si le tuple fait partie d'une grande expression).

Les tuples, comme les chaînes de caractères, sont immuables: ce n'est pas possible de modifier les éléments d'un tuple. Il est possible de créer des tuples contenant des objets mutables, comme des listes.

L'instruction `t = 12345, 54321, 'hello!'` est un exemple d'un *tuple packing*: les valeurs `12345`, `54321` et `'hello!'` sont combinées toutes les trois dans un tuple. L'opération inverse est aussi possible:

```
>>> x, y, z = t
```

Cela est appelé *sequence unpacking* ou déballage de séquence, et fonctionne pour toutes les séquence. Le *sequence unpacking* nécessite que la liste des variables à gauche possède le même nombre d'éléments que dans la séquence.

ENSEMBLES (SETS)

Python inclus aussi un type de données pour les ensemble. Un `set` est une collection non-ordonnée composée uniquement d'éléments différents les uns des autres. Les usages de base comprennent les tests d'appartenance et l'élimination des doublons. Les ensembles supportent également les opérations mathématiques telles l'union, l'intersection, la différence et la différence symétrique.

Voici une courte démonstration:

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> fruit = set(basket)                                     # creation d'un set sans doublons
>>> fruit
set(['orange', 'pear', 'apple', 'banana'])
>>> 'orange' in fruit                                    # rapide test d'appartenance
True
>>> 'crabgrass' in fruit
False
>>> # Démontre les opérations des ensembles sur les lettres uniques de deux mots
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                                 # lettre unique dans a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b                                           # la lettre est dans a mais pas dans b
set(['r', 'd', 'b'])
>>> a | b                                           # la lettre est dans a ou dans b ou dans les deux
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b                                           # la lettre est dans a et dans b
set(['a', 'c'])
>>> a ^ b                                           # la lettre est dans a ou dans b mais pas dans les deux
set(['r', 'd', 'b', 'm', 'z', 'l'])
```

DICTIONNAIRE

Un autre type de données construite dans Python est le dictionnaire. Contrairement aux séquences qui sont indexées par un nombre, les dictionnaires sont indexés par des clés, qui peuvent être n'importe quel type immuable. Les tuples peuvent être utilisés comme des clés s'ils contiennent seulement des chaînes de caractères, des nombres ou des tuples; si un tuple contient n'importe quel objet mutable directement ou indirectement, il ne peut pas être utilisé comme clé. Vous ne pouvez pas utiliser les listes comme des clés puisqu'elles peuvent être modifiées sur place.

Une paire d'accolades crée un dictionnaire vide: `{}`. Placer une liste de paires de `key:value` séparées par une virgule entre accolades ajoute des paires de `key:value` initiales au dictionnaire; c'est également la façon dont le dictionnaire est affiché.

L'opération principale du dictionnaire est l'entreposage de valeurs à l'aide d'une clé et l'extraction de la valeur étant donnée sa clé. Il est également possible de supprimer une paire de `key:value` à l'aide de `del`. Une assignation utilisant une clé existante écrasera la valeur associée précédente. Extraire une valeur à l'aide d'une clé inexiste cause une erreur.

La fonction `keys()` du dictionnaire retourne une liste de toutes les clés utilisées dans le dictionnaire, selon un ordre arbitraire (si vous les voulez dans l'ordre, vous pouvez simplement appliquer la fonction `sorted()`). Pour vérifier qu'une clé est dans le dictionnaire, on utilise le mot-clé `in`.

Voici un petit exemple de l'utilisation d'un dictionnaire:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
```

```
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
```

TECHNIQUES D'ITÉRATION

Lorsqu'on itère parmi les éléments d'un dictionnaire la clé et la valeur correspondante peuvent être retrouvées en même temps grâce à la méthode `iteritems()`.

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.iteritems():
...     print k, v
...
gallahad the pure
robin the brave
```

Lorsqu'on itère parmi les éléments d'une séquence, l'indice et la valeur correspondante peuvent être retrouvés en même temps grâce à la fonction `enumerate()`.

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print i, v
...
0 tic
1 tac
2 toe
```

Pour itérer sur deux séquences ou plus en même temps, les éléments peuvent être combinés à l'aide de la fonction `zip()`.

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print 'What is your {0}?  It is {1}.'.format(q, a)
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

Pour itérer sur les éléments d'une séquence en sens inverse, appelez la fonction `reversed()`.

```
>>> for i in reversed(xrange(1,10,2)):
...     print i
...
9
7
5
3
1
```

Pour itérer sur les éléments d'une séquence ordonnée, utilisez la fonction `sorted()` qui retournera une nouvelle liste ordonnée en laissant la liste originale non-alterée.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print f
...
apple
banana
orange
pear
```

MODULES

Une fois l'interpréteur fermé, les définitions qui ont été faites (fonctions et variables) sont perdues. Évidemment, si vous désirez écrire de plus long programme, il est préférable d'utiliser un éditeur texte pour écrire des scripts et par la suite, les exécuter. Vous pourriez vouloir séparer votre programme en plusieurs fichiers, ou bien réutiliser des fonctions que vous jugez pratiques sans avoir à les recopier dans chacun de vos programmes.

Pour remédier à cela, Python permet de mettre les définitions dans un fichier et de les utiliser à partir d'autres scripts ou dans l'interpréteur. Ces fichiers se nomment *modules*; les définitions d'un module peuvent être importées dans d'autres modules.

Un module est un fichier contenant des expressions et des définitions en Python. Le nom du fichier est le nom du module avec l'extension `.py`. Par exemple, utilisez un éditeur de texte pour créer un fichier nommé `fibo.py` et écrivez-y:

```
# Fibonacci numbers module

def fib(n):      # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Maintenant allez dans l'interpréteur interactif de Python et importez ce module à l'aide de la commande suivante:

```
>>> import fibo
```

En utilisant le nom du module, on peut accéder aux fonctions:

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Si vous planifiez utiliser une fonction souvent, il est possible de l'assigner à une variable locale:

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

EXÉCUTER UN MODULE COMME UN SCRIPT

Lorsque vous exécutez un module Python avec

```
python fibo.py <arguments>
```

le code dans le module sera exécuté, exactement comme si vous l'aviez importé, mais avec la variable globale `__name__` assignée à `"__main__"`. Ce qui implique qu'en ajoutant ce code à la fin du module:

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

il est possible d'utiliser ce fichier en tant que script ainsi qu'en tant que module, puisque le code accédant aux arguments du script ne sera exécuté que si le module est utilisé comme un script:

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

Si le module est importé, le code ne sera pas exécuté:

```
>>> import fibo
>>>
```

SCOPES (PORTÉE)

Python utilise un *lexical scope* (portée lexicale), c.-à-d. que la liste des noms accessibles à un moment de l'exécution est définie par le code source. Pour résoudre à quelle variable un nom fait référence, Python cherche:

1. d'abord localement dans la fonction
2. puis dans les autres fonctions englobantes (de l'intérieur vers l'extérieur)
3. ensuite dans le module contenant la définition de la fonction
4. et finalement cherche parmi les noms intégrés (*built-in*)

```
>>> a = 1
>>> b = 2
>>> c = 3
>>> def blu():
...     b = 4
...     c = 5
...     def bla():
...         c = 6
...         print 'In bla(): a =',a , 'b =', b, 'c =', c
...     bla()
...     print 'In blu(): a =',a , 'b =', b, 'c =', c
...     return bla
...
>>> bla = blu()
In bla(): a = 1 b = 4 c = 6
In blu(): a = 1 b = 4 c = 5
>>> bla()
In bla(): a = 1 b = 4 c = 6
>>> print 'In global: a =',a , 'b =', b, 'c =', c
In global: a = 1 b = 2 c = 3
```

Il est important de noter qu'en Python, bien que la portée soit lexicale, l'évaluation de la valeur associée à un nom est faite de façon **dynamique**. Ainsi, un nom dans une fonction peut changer de valeur si la variable associée change:

```
>>> def blu():
...     b = 4
...     def bla():
...         print 'In bla(): b =', b
...     bla()
...     b = 5
...     return bla
...
>>> bla = blu()
In bla(): b = 4
>>> bla()
In bla(): b = 5
```

L'ESPACE DE RECHERCHE DES MODULES (SEARCH PATH)

Lorsqu'un module nommé `spam` est importé, l'interpréteur cherche un fichier nommé `spam.py` dans le répertoire contenant le script exécuté, puis ensuite dans la liste des répertoires spécifiés par la variable d'environnement `PYTHONPATH`.

FICHIERS PYTHON COMPLIÉS

Une optimisation utile du temps de démarrage des programmes utilisant de nombreux modules est faite automatiquement par Python, via la création de versions compilées en *byte-code* de ces modules. Si un fichier nommé `spam.pyc` existe dans le répertoire où `spam.py` se situe, alors il est supposé que celui-ci correspond à sa version compilée. Cependant, si une modification à `spam.py` a été apportée depuis la création de `spam.pyc`, ce dernier sera ignoré.

LA FONCTION `DIR()`

La fonction intégrée `dir()` est utilisée pour connaître le nom des définitions qu'un module contient. Elle retourne une liste triée de chaînes de caractères:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
```

Sans arguments, `dir()` liste les noms définis actuellement:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__doc__', '__file__', '__name__', 'a', 'fib', 'fibo', 'sys']
```

Notez qu'il énumère tous les types de noms: les variables, les modules, les fonctions, etc.

PACKAGES

Les *packages* sont une façon pratique de structurer les modules en utilisant “*dotted module names*”. Par exemple, ce nom de module `A.B` désigne un sous-module nommé `B` inclus dans le *package* nommé `A`.

Supposons que l'on désire créer une collection de modules (un “*package*”) afin d'uniformiser la gestion de fichiers sonores. Une structure possible du *package* (exprimé en terme de structure de système de fichiers hiérarchique) serait:

```
sound/
    __init__.py           Top-level package
    formats/
        __init__.py       Initialize the sound package
        wavread.py        Subpackage for file format conversions
        wavwrite.py
        aiffread.py
        aiffwrite.py
        auread.py
        auwrite.py
        ...
    effects/              Subpackage for sound effects
        __init__.py
        echo.py
        surround.py
        reverse.py
        ...
    filters/              Subpackage for filters
        __init__.py
        equalizer.py
        vocoder.py
        karaoke.py
        ...
```

Lorsqu'un *package* est importé, Python cherche parmi les répertoires spécifiés par le `sys.path`, le sous-répertoire du *package*.

Les fichiers `__init__.py` indiquent à Python que ces répertoires contiennent des *packages*. Dans le plus simple des cas, `__init__.py` peut être un fichier vide, mais il peut également exécuter du code d'initialisation pour le *package*.

Les utilisateurs des *packages* peuvent importer les modules individuellement à partir d'un *package*. Par exemple:

```
import sound.effects.echo
```

Ceci charge le sous-module `sound.effects.echo`. Par la suite, il devra être référencé par son nom entier.

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

Une alternative est d'importer le module comme suit:

```
from sound.effects import echo
```

Ceci charge également le sous-module `echo`, mais il pourra être référencé sans spécifier le nom du *package* en préfix et pourra donc être utilisé de cette façon:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Une autre variation est d'importer directement la fonction ou la variable désirée:

```
from sound.effects.echo import echofilter
```

Encore une fois, cela charge le sous-module `echo`, mais rend la fonction `echofilter()` directement disponible:

```
echofilter(input, output, delay=0.7, atten=4)
```

LIBRAIRIE STANDARD DE PYTHON

Python vient avec plusieurs modules intégrés (*built-in*), qui font partie de la librairie standard. La librairie standard inclue également d'autres types, fonctions et structures de données qui ne sont pas décrites dans ce tutoriel.

La librairie standard se trouve à l'adresse suivante: <http://docs.python.org/release/2.6.5/library/index.html>. Il est fortement recommandé de la consulter avant d'implémenter soi-même toute fonction ou structure de données nécessaire, puisqu'elle pourrait déjà être présente dans la librairie.

LECTURE ET ÉCRITURE DE FICHIERS

La fonction `open()` retourne un objet fichier et accepte deux arguments: `open(filename, mode)`.

```
>>> f = open('/tmp/workfile', 'w')
>>> print f
<open file '/tmp/workfile', mode 'w' at 80a0960>
```

Le premier argument est une chaîne de caractère contenant le nom du fichier. Le second argument est une autre chaîne contenant quelques caractères décrivant la façon dont le fichier sera utilisé. `mode` peut être `'r'` afin d'ouvrir le fichier en lecture seule, `'w'` afin d'ouvrir le fichier pour écriture seulement (un fichier existant avec le même nom sera écrasé), ou `'a'` qui permet d'ouvrir le fichier pour y insérer des caractères à la fin. `'r+'` permet d'ouvrir le fichier en lecture et en écriture. L'argument `mode` est optionnel et par défaut le fichier est ouvert en lecture seule.

MÉTHODES DES OBJETS FICHIER

Tous les exemples de cette section suppose qu'un objet fichier nommé `f` existe.

Pour lire le contenu d'un fichier, on utilise une des fonctions suivantes: `f.read([size])`, `f.readline()` ou `f.readlines()`.

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''

>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''

>>> f.readlines()
['This is the first line of the file.\n', 'Second line of the file\n']
```

Une approche alternative est de lire les lignes en itérant l'objet fichier. Ceci est simple et performant tant en mémoire qu'en rapidité:

```
>>> for line in f:
...     print line,
This is the first line of the file.
Second line of the file
```

La fonction `f.write(string)` écrit le contenu d'une chaîne de caractères dans le fichier.

```
>>> f.write('This is a test\n')
```

Afin d'écrire d'autres types de données, il faut d'abord les convertir en chaîne de caractères:

```
>>> value = ('the answer', 42)
>>> s = str(value)
>>> f.write(s)
```

Lorsque vous avez terminé avec un fichier, appelez la fonction `f.close()` pour fermer le fichier et libérer les ressources système. Après avoir appelé `f.close()`, toutes tentatives d'utiliser l'objet fichier échouera.

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

LE MODULE PICKLE

Les chaînes de caractères peuvent être facilement écrites et lues à partir d'un fichier. En ce qui concerne les nombres, c'est un peu plus compliqué puisque la méthode `read()` retourne seulement des chaînes de caractères qui devront être passées à une fonction comme `int()`, qui s'occupera de convertir, par exemple, `'123'` en valeur numérique 123. Par contre, lorsqu'on veut sauvegarder des types de données plus complexes comme des listes, dictionnaires, voir même des instances de classe, les choses se compliquent.

Si vous avez un objet `x`, et un objet fichier `f` qui a été ouvert en écriture, il suffit d'utiliser le module `pickle` pour sauvegarder l'objet:

```
pickle.dump(x, f)
```

L'opération inverse est tout aussi simple, si `f` est un objet fichier qui a été ouvert en lecture:

```
x = pickle.load(f)
```

CLASSES

DÉFINITION D'UNE CLASSE

La forme la plus simple pour définir une classe ressemble à:

```
class ClassName:  
    <statement-1>  
    .  
    .  
    .  
    <statement-N>
```

Les définitions de classes ressemblent aux définitions de fonctions (mot-clé `def`).

Les instructions à l'intérieur de la définition d'une classe vont être des définitions de fonctions et de variables.

Lorsqu'on exécute la définition d'une classe, un nouveau *namespace* est créé et sert de *scope* local. Donc, les définitions de fonctions et de variables iront dans ce *namespace*.

OBJETS D'UNE CLASSE

Les objets d'une classe supporte deux sortes d'opérations: référencement aux attributs (variables et fonctions) et l'instantiation.

Le *référencement aux attributs* utilise la syntaxe Python standard: `obj.name`. Les attributs valides sont tous les noms (variables et fonctions) définis lorsque la classe a été créée. Ainsi, si nous avons une définition de classe qui ressemble à:

```
class MyClass:  
    """Un exemple simple de classe"""  
    i = 12345  
    def f(self):  
        return 'hello world'
```

alors `MyClass.i` et `MyClass.f` sont des attributs valides, qui retournent respectivement un entier et un objet fonction. Les attributs de la classe peuvent également être modifiés, ainsi `MyClass.i` peut être modifié en utilisant l'affectation.

L'*instantiation* d'une classe utilise la notation des appels de fonctions. Par exemple:

```
x = MyClass()
```

crée une nouvelle instance de la classe et assigne l'objet résultant à une variable locale `x`.

L'opération d'instantiation crée un objet vide. Or, il est possible de créer des instances ayant un état initial. Pour y arriver, la classe peut définir une méthode spéciale nommée `__init__()`, comme ceci:

```
def __init__(self):  
    self.data = []
```

Si une classe défini la méthode `__init__()`, alors lors de l'instantiation la méthode `__init__()` est automatiquement appelée. Donc, dans l'exemple, une nouvelle instance initialisée peut être obtenue en faisant:

```
x = MyClass()
```

Bien sûr, la méthode `__init__()` peut prendre des arguments. Par exemple,

```
>>> class Complex:  
...     def __init__(self, realpart, imagpart):
```

```
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

Habituellement, une méthode peut être appelée immédiatement après l'instantiation:

```
x.f()
```

Dans l'exemple avec la classe `MyClass`, la méthode retournera la chaîne de caractères `'hello world'`.

Vous avez sans doute remarqué que `x.f()` a été appelée sans spécifier d'arguments bien que la définition de la méthode `f()` spécifie un paramètre! En fait, l'objet qui a servi à l'appel de fonction est passé automatiquement comme argument. Dans notre exemple, l'appel `x.f()` est exactement équivalent à `MyClass.f(x)`.

REMARQUES

Les variables membres écrasent (*override*) les méthodes qui ont le même nom; pour éviter ces conflits, on utilise des conventions de noms (verbes pour les méthodes, minuscule-majuscule, etc.).

Dans une méthode, il est possible d'en appeler d'autres en utilisant l'argument `self`:

```
class Bag:
    def __init__(self):
        self.data = []
    def add(self, x):
        self.data.append(x)
    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

L'HÉRITAGE

La syntaxe pour définir une classe dérivée ressemble à:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Le nom `BaseClassName` doit être défini dans le *scope* contenant la définition de la classe dérivée (c.-à-d. doit être accessible). Sinon, on peut spécifier le nom du module:

```
class DerivedClassName(modname.BaseClassName):
```

La classe dérivée peut redéfinir (*override*) des méthodes de la classe de base. (Pour les programmeurs C++: toutes méthodes en Python sont `virtual`.)

Il est possible d'appeler la méthode de base en appelant: `BaseClassName.methodname(self, arguments)`.

Python possède deux fonctions intégrées qui sont utiles lorsqu'on travaille avec l'héritage:

- Utilisez `isinstance()` pour vérifier le type (le nom de la classe) de l'instance: `isinstance(obj, int)` sera `True` seulement si `obj.__class__` est un `int` ou bien une classe qui hérite de `int`.
- Utilisez `issubclass()` pour vérifier l'héritage d'une classe: `issubclass(bool, int)` est `True` puisque `bool` dérive de `int`.

VARIABLE PRIVÉE

En Python, le concept de variable privée n'existe pas. Par contre, il existe une convention pour le simulé: un nom (de variable ou de méthode) qui est préfixé avec un *underscore* (ex. `_spam`) devrait être considéré comme étant privé dans son usage.

EN PYTHON, IL N'Y A PAS DE POINTEURS

Toutes les variables en Python sont des références ou alias à des objets; il n'existe pas de notion de pointeur, comme en C++. Par contre, ceci n'empêche pas deux variables de référer au même objet en mémoire. En fait, l'assignation d'instances

de classes à des variables ne crée pas de copies des instances. Ces variables pointeront plutôt vers le même espace mémoire:

```
>>> a = Bag()
>>> b = a
>>> b.add('hello')
>>> b.data
['hello']
>>> a.data
['hello']
>>> a == b
True
```

Par défaut, l'opérateur `==` vérifie si les variables font références à la même instance de classe (on pourrait dire "pointent" vers le même objet).

NUMPY

Numpy est un package Python permettant de manipuler efficacement des tableaux multidimensionnels. Il est très utilisé pour représenter des vecteurs/matrices et leur appliquer des opérations mathématiques de façon très efficace.

LE TABLEAU (`NDARRAY`)

Le type d'objet principal en Numpy est le tableau multidimensionnel: `numpy.array` alias `ndarray`. C'est une table d'éléments (habituellement des nombres), qui sont tous du même type et qui sont indexés par un tuple (plusieurs dimensions) d'entiers positifs.

Notez que dans la librairie Numpy, une dimension est également appelée axe (*axis*).

CRÉATION DE TABLEAUX

Il y a plusieurs façons de créer des tableaux.

La première consiste à créer un tableau à partir d'une liste ou d'un tuple en utilisant la fonction `array`.

```
>>> from numpy import *
>>> a = array([2,3,4])
>>> a
array([2, 3, 4])
>>> a.dtype
dtype('int32')
>>> b = array([1.2, 3.5, 5.1])
>>> b.dtype
dtype('float64')
```

Une erreur fréquente est d'omettre les crochets, ce qui revient à appeler `array` avec plusieurs arguments au lieu d'une séquence de nombres.

```
>>> a = array(1,2,3,4)      # Erreur
>>> a = array([1,2,3,4])   # OK
```

`array` transforme les séquences imbriquées en tableau multidimensionnel. Par exemple:

```
>>> b = array([(1.5,2,3), (4,5,6)])
>>> b
array([[ 1.5,    2. ,    3. ],
       [ 4. ,    5. ,    6. ]])
```

Le type de données peut également être spécifié à la création. Par défaut, il est automatiquement détecté. Ici, on force les éléments du tableau à être des nombres complexes:

```
>>> c = array([ [1,2], [3,4] ], dtype=complex )
>>> c
array([[ 1.+0.j,  2.+0.j],
       [ 3.+0.j,  4.+0.j]])
```

Parfois, les éléments que l'on désire conserver dans un tableau ne sont pas connus au départ, mais le nombre d'éléments est connu. Dans ce cas, il existe quelques fonctions permettant de créer des tableaux avec l'espace nécessaire préalablement réservé, ce qui minimise la réallocation à chaque agrandissement du tableau.

La fonction `zeros` permet de créer un tableau contenant seulement des zéros; la fonction `ones` crée une tableau avec des uns; et la fonction `empty` crée un tableau rempli d'éléments indéterminés (c.-à-d. avec ce qu'il y avait déjà en mémoire). Tout ce qu'il faut, c'est de spécifier les dimensions du tableau. Voici un exemple:

```
>>> zeros( (3,4) )
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
>>> ones( (2,3,4), dtype=int16 )           # dtype peut aussi être spécifié
array([[[1, 1, 1, 1],
       [1, 1, 1, 1],
       [1, 1, 1, 1]],
      [[1, 1, 1, 1],
       [1, 1, 1, 1],
       [1, 1, 1, 1]]], dtype=int16)
>>> empty( (2,3) )
array([[ 3.73603959e-262,   6.02658058e-154,   6.55490914e-260],
       [-5.30498948e-313,   3.14673309e-307,   1.00000000e+000]])
```

ARANGE ET Linspace

Numpy offre deux fonctions facilitant la création de séquences de nombres. Identique à la fonction `range` à l'exception qu'elle retourne un `array` au lieu d'un `list`, `arange` prend en argument le nombre de départ, le nombre de fin et le pas (`step`).

```
>>> arange( 10, 30, 5 )
array([10, 15, 20, 25])
>>> arange( 0, 2, 0.3 )                 # les réels sont acceptés
array([ 0. ,  0.3,  0.6,  0.9,  1.2,  1.5,  1.8])
```

Lorsque l'on désire créer des séquences de nombres réels et que l'on connaît le nombre d'éléments dont on a besoin, la fonction `linspace` est pratique. `linspace` prend en argument le nombre d'éléments désirés plutôt que le pas (`step`).

```
>>> linspace( 0, 2, 9 )                # 9 nombres de 0 à 2
array([ 0. ,  0.25,  0.5 ,  0.75,  1. ,  1.25,  1.5 ,  1.75,  2. ])
>>> x = linspace( 0, 2*pi, 100 )        # utile pour évaluer une fonction pour des points donnés
>>> f = sin(x)
```

QUELQUES PROPRIÉTÉS DES TABLEAUX (ARRAYS)

- `ndarray.ndim`: le nombre de dimensions.
- `ndarray.shape`: les dimensions du tableau contenues dans un tuple de taille égale à `ndarray.ndim`.
- `ndarray.size`: nombre total d'éléments dans le tableau et donc égal au produit des éléments de `ndarray.shape`
- `ndarray.dtype`: le type des éléments du tableau (ex. `int`, `float`, `numpy.int32`, `numpy.int16`, and `numpy.float64`, etc)

```
>>> from numpy import *
>>> a = numpy.array([[0, 1, 2, 3, 4],
                   [5, 6, 7, 8, 9]])
>>> a.shape
(2, 5)
>>> a.ndim
2
>>> a.dtype.name
'int32'
>>> a.size
10
>>> type(a)
numpy.ndarray
```

LES OPÉRATEURS DE BASE

ARITHMÉTIQUE

Les opérateurs d'arithmétique sont appliqués élément par élément (*element-wise*). Le résultat est contenu dans un nouveau tableau de même dimension. Par exemple:

```
>>> a = array([20,30,40,50])
>>> b = arange( 4 )
>>> b
array([0, 1, 2, 3])
>>> c = a-b
>>> c
array([20, 29, 38, 47])
```