Python

Réprise du tutoriel de Daniel Nouri

23 mai 2009

Mode interactif

- Tapez “python“ dans la ligne de commande. Vous devez obtenir :

```
Python 2.6 (#1, Feb 28 2007, 00:02:06)
Type "help", "copyright", "credits" or "license"
for more information.
>>>
```

- Pour sortir tapez “Ctrl-Z“ ou “Ctrl-D“

Notre premier programme

```
>>> world_is_flat = 1
>>> if world_is_flat:
...     print "Don't fall off!"
...
Don't fall off!
```

Utilisez Python comme calculatrice :

```
>>> 2+2
    4
```

Les opérateurs “+“, “-“, “*“, “/“, “(“ et “)“ fonctionnent comme les autres langages

```
>>> 50-5*6
    20
>>> (50-5)*6
    270
```

La division des entiers renvoie la partie entière

```
>>> 7/3
    2
>>> 7/-3
    -3
```

Le signe égal ("=") attribue une valeur à une variable

```
>>> width = 20
>>> height = 5*9
>>> width * height
900
```

Operations en virgule flotante

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

La derniere expression affichée est attribuée à la variable “_“

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

Chaînes de caractères

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

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

Les opérateurs “+“ et “*“

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

Indexation des mots.

Le type caractère n'existe pas séparément

```
>>> word
'HelpA'
>>> word[4]
'A'
```

Sous-chaînes en utilisant le *découpage*

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

```
>>> word[:2] # les 2 premiers caractères
'He'
>>> word[2:] # Tout sauf les 2 premiers
...          # caractères
'lpA'
```

Les chaînes Python sont *immutables*

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

Par contre la création des nouvelles chaînes est facile et efficiente

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

Le dépassement des indices est bien capté

```
>>> word[1:100]
'elpA'
>>> word[10:]
''
>>> word[2:1]
''
```

Les indexes peuvent être des numéros négatifs, pour commencer à compter de droite

```
>>> word[-1] # le dernier caractère
'A'
>>> word[-2] # le avant-dernier caractère
'p'
```

```
>>> word[-2:] # les 2 derniers caractères
'pA'
>>> word[:-2] # tout sauf les 2 derniers caractères
'Hel'
>>> word[-100:] # L'index dépassé est coupé
'HelpA'
```

La fonction prédéfinie “len“ renvoie la longueur de la chaîne

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

Listes

- un de type *composé* de données de Python
- peuvent être représentées comme une liste des valeurs(items) séparées par virgule entre les crochets
- les items peuvent ne pas être du même type

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

Comme les chaînes, les listes supportent les opération des index

```
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
```

Les listes peuvent être découpées et concaténées

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

Différemment des chaînes, il est possible de changer des éléments individuels de la liste

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

Des attributions aux sections sont aussi possibles

```
>>> # Remplace quelques items:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Supprime quelques uns:
... a[0:2] = []
>>> a
[123, 1234]
```

```
>>> # Insere quelques uns:
... a[1:1] = ['bletch', 'xyzzy']
>>> a
[123, 'bletch', 'xyzzy', 1234]
>>> # Insere (un copie de) elle-même:
... a[:0] = a
>>> a
[123, 'bletch', 'xyzzy', 1234, 123, 'bletch',
'xyzzy', 1234]
```

La fonction prédéfinie “len“ fonctionne aussi avec les listes

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

Il est possible d'imbriquer des listes

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
```

```
>>> p[1].append('xtra')
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

Notez que “p[1]“ et “q“ référencent le même objet !

Premiers pas vers la programmation

- *Attribution multiple* : les expressions à droite sont toutes évaluées avant que l'attribution s'effectue.
- La boucle “while“ s'exécute temps que la condition est vraie.
- Les opérateurs de comparaison fonctionnent comme en C : “¡“, “¿“, “==“, “¡=“, “¿=“, “!=“.
- Toute valeur non-nulle est vraie, comme en C. Zéro est faux.
- La condition doit être une chaîne ou liste ; en fait n'importe quelle séquence.
  Une séquence vide est fausse, non-vide est vraie.

- Le *corps* de la boucle est *indenté* : l'indentation permettant en Python de grouper les blocks.
  En pratique votre éditeur vous aidera avec l'indentation.
- La fonction “print“ affiche la valeur de l'expression fourni en argument.

“If“

```
>>> x = 42
>>> if x < 0:
...         x = 0
...         print 'Négatif changé à zéro'
... elif x == 0:
...         print 'Zéro'
... elif x == 1:
...         print 'Un'
... else:
...         print 'Plusieurs'
etc
```

- On peut avoir aucune ou plusieurs parties “elif“ et la partie “else“ est optionelle.
- “elif“ est l’abréviation de *else if*
- La séquence “if“ ... “elif“ ... “elif“ ... remplace les séquences “switch“ ou “case“ trouvées dans autres languages.

“For“

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

- Peut être différent de ce que vous êtes habitués.
- "for" itère dans tous les items de la séquence dans l'ordre qu'ils apparaissent.

Il n'est pas prudent de modifier la séquence en l'itérant. Faites une copie si vous en avez besoin

```
>>> for x in a[:]: # faites une copie de la
                   séquence
...     if len(x) > 6: a.insert(0, x)
...
>>> a
['defenestrate', 'cat', 'window', 'defenestrate']
```

“break“ et “continue“

- “break“ arrête la plus petite boucle “for“ ou “while“.
- “continue“ continue avec l'itération suivante de la boucle.
- c'est tout, ces déclarations fonctionnent comme en C.

```
<<< a
['defenestrate', 'cat', 'window', 'defenestrate']
<<< f = raw_input("Find: ")
Find: window
<<< for x in a:
...     if f == x:
...         print "Found", f
...         break
Found window
```

Les clauses “else“ dans les boucles

Les boucles peuvent avoir un “else“ qui est executé quand la boucle *n'est pas* terminée par un “break“.

```
>>> f = 'dog'
>>> for x in a:
...     if f == x:
...         print "Found", f
...         break
... else:
...     print f, "not found"
dog not found
```

Fonctions

```
>>> def replace(l, fro, to):
...     for i in range(len(l)):
...         if l[i] == fro:
...             l[i] = to

>>> k = ['one', 'two']
>>> replace(k, 'one', 'three')
>>> k
['three', 'two']
```

- Le mot clé “def“ introduit une définition d'une fonction, suivi par le nom de la fonction et les paramètres entre les parenthèses.
- Le corps de la fonction commence sur la ligne suivante indententé.
- Les valeurs en Python sont *toujours* des références objet.
- Définition d'une fonction introduit le nom dans la table des symboles.
- La fonction peut être attribuée à un autre nom

```
>>> r = replace
>>> r(k, 'three', 1)
>>> k
[1, 'two']
```

Utilisez le mot clé “return“ pour retourner une valeur d'une fonction

```
>>> def absadd(a, b):
...     return abs(a + b)
>>> absadd(-5, -10)
15
>>> a = absadd(absadd(-5, -10), -100)
>>> a
85
```

Arguments de la fonction

Il est possible de définir des fonctions avec un nombre variable d'arguments, comme "range"

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

Les *valeurs par défaut des arguments* sont très utiles

```
>>> def replace3(l, fro, to, count=0):
...     l = l[:]
...     for i in range(len(l)):
...         if l[i] == fro:
...             l[i] = to
...             count = count - 1
...             if count == 0: break
...     return l
```

```
>>> replace3(['one', 'two', 'one', 'two'],
...          'one', 'three')
['three', 'two', 'three', 'two']
>>> replace3(['one', 'two', 'one', 'two'],
...          'one', 'three', 1)
['three', 'two', 'one', 'two']
```

Comment *provoquer une exception* ?

```
>>> input = 'N'
>>> if input == 'yes':
...     value = True
... elif input == 'no':
...     value = False
... else:
...     raise ValueError("Invalid input!")
Traceback (most recent call last):
ValueError: Invalid input!
```

Les listes avec des arguments arbitraires sont moins fréquentes

```
>>> def cheeseshop(kind, *arguments, **keywords):
...     print "-- Do you have any", kind, "?"
...     print "-- I'm sorry, we're all out of", kin
...     for arg in arguments:
...         print arg
...     print "-" * 40
...     keys = keywords.keys()
...     keys.sort()
...     for kw in keys: print kw, ":", keywords[kw]
```

```
>>> cheeseshop("Limburger", "It's very runny, sir."
...            "It's really very, VERY runny, sir."
...            shopkeeper='Michael Palin',
...            client="John Cleese",
...            sketch="Cheese Shop Sketch")
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch
```

Si vos arguments sont déjà une liste(ou un tuple), vous pouvez utiliser l'opérateur "*" pour les décompresser

```
>>> range(3, 6) # appel habituel
[3, 4, 5]
>>> args = [3, 6]
>>> range(*args) # arguments décompressés
[3, 4, 5]
```

Pareil pour les dictionnaires

```
>>> def parrot(voltage, state='a stiff', action='vo
...     print "-- This parrot wouldn't", action,
...     print "if you put", voltage, "volts through
...     print "E's", state, "!"
>>> d = {"voltage": "four million",
...      "state": "bleedin' demised",
...      "action": "VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four millio
```

```
>>> range2(5)
[0, 1, 2, 3, 4]
>>> range2(1, 5)
[1, 2, 3, 4]
>>> range2(1, 6, 2)
[1, 3, 5]
```

Trouver la documentation

Python contien la documentation integrée.
Essayez “help(max)“ ou “help(filter)“

Comment savoir quelles méthodes a une liste ?
Souvenez-vous de “l.append“.
Quelles autre méthodes existent ? Essayez “dir(l)“.
Pour chaque méthodes vous pouvez exécuter “help(méthode)“, par ex. “help(l.insert)“.
Vous pouvez aussi exécuter “help(l)“.

Lambda

Avec “lambda“ de petites fonctions anonymes peuvent être crées.
Cette fonction

```
>>> lambda a, b: a + b
<function <lambda> at ...>
```

est équivalente à celle-ci :

```
>>> def add(a, b):
...     return a + b
```

On peut aussi lui attribuer un nom

```
>>> add = lambda a, b: a + b
>>> add(5, -3)
2
```

Fichiers

“open“ renvoie un objet fichier et est d’habitude utilisé avec 2 argumnets : “open(filename, mode)“

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

Le second argument contient une chaîne décrivant le mode d'ouverture du fichier

- “'r'“ pour lecture seule
- “'w'“ pour écriture seule
- “'a'“ pour ajout
- “r+“ pour lecture-écriture

Sous Windows, “’b’“ ajouté au mode d’ouverture ouvre le fichier en mode binaire.
Windows fait la différence entre les fichiers text et binaires ; les fins de lignes dans les fichiers text sont automatiquement enlevées quand les données sont écrites, lues.

Écriture dans un fichier

```
>>> f.write('This is the first.\n')
>>> f.close() # flush
```

Lecture

```
>>> f = open('/tmp/workfile') # 'r' default
>>> f.read() # read the entire file
'This is the first.\n'
>>> f.close()
```

Lecture ligne par ligne

```
>>> f = open('/tmp/workfile', 'r')
>>> for line in f:
...         print line, # includes \n
This is the first.
This is the second,
and this is the third line.
>>> f.close()
```

Encodage des chaînes de caractères

```
>>> 'Hello, Rabat!'.encode('base64')
'SGVsbG8sIFJhYmF0IQ==\n'
>>> _.decode('base64')
'Hello, Rabat!'
```

Tuples

Les tuples sont comme les listes, sauf qu'ils sont immutables, comme les chaînes

```
>>> t = (12345, 54321, 'hello!')
>>> t[0]
12345
>>> t[0] = 67890
Traceback (most recent call last):
...
TypeError: 'tuple' object does not support item ass
```

Une belle fonctionnalité est le *dépaquetage séquentiel*

```
>>> x, y, z = t
>>> x
12345
>>> y
54321
>>> z
'hello!'
```

Sets

Un set c'est une collection non-ordonnée avec des elements uniques

```
>>> basket = ['apple', 'orange', 'apple',
...           'pear', 'orange', 'banana']
>>> fruit = set(basket) # no duplicates
>>> fruit
set(['orange', 'pear', 'apple', 'banana'])
```

```
>>> 'orange' in fruit # fast test
True
>>> 'crabgrass' in fruit
False
>>> fruit.add('strawberry')
>>> len(fruit)
5
```

Opérations avec les lettres de mots

```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a # unique letters in a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b # letters in a but not in b
set(['r', 'b', 'd'])
```

```
>>> union = a | b
>>> intersection = a & b
>>> xor = a ^ b
```

Dictionnaires

Un dictionnaire est un ensemble non-ordonnées de paires *clé* :*valeur* à condition que la clé soit unique.
Une paire d'accolades crée un dictionnaire vide : "{}".
Tout type immutable peut être utilisé comme clé, comme les chaînes, les tuples. Les listes peuvent pas être utilisées comme clés, sauf si elles peuvent être modifiées sur place
Les opérations principales sont :

- enregistrement d'un valeur avec une clé
- extraction d'une valeur par sa clé

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

```
>>> tel
{'sape': 4139, 'jack': 4098, 'guido': 4127}
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'irv': 4127, 'guido': 4127}
>>> tel.keys()
['jack', 'irv', 'guido']
>>> 'guido' in tel
True
```

Zen of Python

```
>>> import this
The Zen of Python, by Tim Peters...
```

Les classes

```
>>> yussuf = {'firstname': 'Yussuf',
...           'lastname': 'Islam'}
>>> def fullname(record):
...     value = record['firstname']
...     middlename = record.get('middlename')
...     if middlename:
...         value += ' ' + middlename
...     value += ' ' + record['lastname']
...     return value
```

```
>>> fullname(yussuf)
'Yussuf Islam'
>>> michael = {'firstname': 'Michael',
...            'lastname': 'Palin',
...            'middlename': 'Abraham'}
>>> fullname(michael)
'Michael Abraham Palin'
```

```
>>> class Person:
...     def __init__(self, firstname, lastname,
...                  middlename=''):
...         self.firstname = firstname
...         self.lastname = lastname
...         self.middlename = middlename
...     def fullname(self):
...         value = self.firstname
...         if self.middlename:
...             value += ' ' + self.middlename
...         value += ' ' + self.lastname
...         return value
```

```
>>> me = Person('Daniel', 'Nouri')
>>> john = Person('John', 'Cleese')
>>> michael = Person('Michael', 'Palin',
...                  'Abraham')
```

```
>>> me.firstname
'Daniel'
>>> me.middlename
''
>>> me.fullname()
'Daniel Nouri'
>>> michael.fullname()
```