Python

Bases du langage

Pierre Navaro

IRMA Strasbourg

IMFS le 28 juin 2011

Présentation de Python

- Projet initié par Guido Von Rossum au début des années 90
- C'est un langage interprété écrit en C.
- Très largement utilisé dans de nombreux domaines.
- Possède peu de limites.
- Syntaxe claire et très simple.
- Les types python sont nombreux et puissants.
- S'interface facilement avec un très grand nombre de langages.
- Permet d'écrire des programmes concis avec un haut niveau d'abstraction.
- Disponible sur Unix, Windows, Mac OS X...

Les différentes versions

- Python 3.x n'est pas une simple amélioration ou extension de Python 2.x.
- Tant que les auteurs de librairies n'auront pas effectué la migration, les deux versions devront coexister.
- Nous nous intéresserons uniquement à Python 2.x car numpy n'existe pas pour la version 3.
- Il existe un grand nombre d'IDE : spyder, eclipse+pydev,netbeans,Wing IDE...
- En plus des versions fourni dans les packages linux, il existe des distributions complètes :
  - SAGEMATH `http://sagemath.org/`
  - PYTHONXY `http://pythonxy.com/`
  - Enthought Python Distribution `http://www.enthought.com/products/epd.php`

Performance

- Python n'est pas rapide ... mais :
  - Ce n'est pas toujours vrai.
  - Certains aspects de Python ont été optimisés.
  - Il y a des modules performants en C/C++/Fortran. (Numpy)
  - C'est le programmeur qui devient plus efficace.
  - Ce qu'il faut optimiser, c'est le temps jusqu'au bout du projet.

  Conseils de programmation
  - Ecrivez votre programme en Python d'abord.
  - Si c'est assez rapide, soyez contents.
  - Sinon optimisez les parties critiques (et rien d'autre)
- L'optimisation trop précoce est souvent source d'erreurs.

Optimisation

- Profilage :

  ```
  python -m profile -s time mon_script.py
  ```

- Améliorer votre algorithme
- Utiliser les modules optimisés (numpy, scipy)
- Tournez vous vers :
  - Pyrex, Cython,
  - C / Swig,
  - C++ / Swig
  - C++ / Boost,
  - Fortran / f2py.

Ecriture d'un script python

test.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
a=2
a
print type(a), a
```

Exécution

```
$ python test.py
<type 'int'> 2
```

Dans une console ipython

```
$ipython
In [1]: run test.py
<type 'int'> 2
```

Les types de données

En Python tout est "objet", il existe d'une vaste gamme de type intégrés. Le typage est dit "dynamique fort".

Commande **type**

```
>>> s = "CEThop"; print type(s)
<type 'str'>
>>> s = [1,2,3,4]; print type(s)
<type 'list'>
>>> s = (1,2,3,4); print type(s)
<type 'tuple'>
>>> s = int(2010); print type(s)
<type 'int'>
>>> s = 3.14; print type(s)
<type 'float'>
```

Les opérateurs

Affectation et opérations de base

```
>>> x = 45
>>> x+2
47
>>> y = 2.5
>>> x+y
47.5
>>> (x*10)/y
180.0
>>>_ % 8 # reste de la division de 180 par 8
4
```

Si vous mélangez les types entiers et réels, le résultat sera un réel.

Opérateurs logiques

```
and or not
```

Opérateurs de comparaison

```
==, is, !=, is not, >, >=, <, <=
```

Les Chaînes de caractères

```
>>> chaine = " IMFS "
>>> chaine += " Python "
>>> 3 * chaine
' IMFS  Python  IMFS  Python  IMFS  Python '
```

Quelques méthodes disponibles : **help(str)**

- **len(s)** : renvoie la taille d'une chaîne,
- **s.find()** : recherche une sous-chaîne dans la chaîne,
- **s.rstrip()** : enlève les espaces de fin,
- **s.replace()** : remplace une chaîne par une autre,
- **s.split()** : découpe une chaîne,
- **s.isdigit()** : renvoie True si la chaîne ne contient que des nombres, False sinon,

Listes, n-uplets et séquences

- Une liste est délimitée par des "[", les n-uplets par des "(".
- Les listes sont "mutables" au contraire des n-uplets.
- **len(liste)** donne la longueur de la liste.
- **list.append()** ajoute un élément.
- Les listes et n-uplets sont indexés à partir de 0.
- Pour parcourir ces listes, on utilise la fonction **range** qui permet d'obtenir une liste d'entiers successifs.
- Les séquences sont des objets dont dérivent les listes, les n-uplets, les chaînes etc....

Pour afficher les méthodes de la classe "liste" :

```
>>> help(list)
```

Accéder aux éléments d'une liste

Slicing [debut : fin : pas]

```
>>> a = 'Demonstrate slicing in Python'.split()
>>> print 'Taille de a = ' + str(len(a))
Taille de a = 4
>>> a.append("2")
>>> print a[-1]
2
>>> print a[:-1]
['Demonstrate', 'slicing', 'in', 'Python']
>>> print a[:]
['Demonstrate', 'slicing', 'in', 'Python', '2']
>>> print a[2:]
['in', 'Python', '2']
>>> print a[-1:]
['2']
>>> print a[-2:]
['Python', '2']
>>> print a[1:3]
['slicing', 'in']
>>> print a[::2]
['Demonstrate', 'in', '2']
```

Accéder aux éléments d'une liste (suite)

Affectation, inversion et tri

```
>>> a = range(5); print a
[0, 1, 2, 3, 4]
>>> b = a   #ici b est une reference de a pas une copie
>>> b[1]= 20; a
[0, 20, 2, 3, 4]
>>> b = a[:3] #dans ce cas il y a copie
>>> b[2]=10; a
[0, 20, 2, 3, 4]
>>> b
[0, 20, 10]
>>> a.reverse(); a
[4, 3, 2, 20, 0]
>>> a.sort(); a
[0, 2, 3, 4, 20]
>>> dir(a)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', ...
>>> id(a)
2852424
```

Un mot sur les dictionnaires

- On les appelle aussi "tableaux associatifs"
- Les indices sont appelés "clés".
- `d['Nom']` affiche l'élément dont la clé est 'Nom'
- `d.keys()` affiche les clés
- `d.items()` affiche les éléments du dictionnaire

```
animal1 = {}
animal1['nom'] = 'girafe'
animal1['taille'] = 5.0
animal1['poids'] = 1100
>>> animal1['nom']='girafe'
>>> animal1['taille'] = 5.
>>> anima1['poids']=1100
>>> print animal1
{'nom': 'girafe', 'poids': 1100, 'taille': 5.0}
>>> print animal1.keys()
['nom', 'poids', 'taille']
>>> print animal1.items()
[('nom', 'girafe'), ('poids', 1100), ('taille', 5.0)]
```

Boucles

La fonction range

```
>>> print range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> print range(5, 10)
[5, 6, 7, 8, 9]
>>> print range(0, 10, 3)
[0, 3, 6, 9]
>>> print range(-10, -100, -30)
[-10, -40, -70]
```

c'est l'indentation qui délimite le corps de la boucle.

Boucle for

```
>>> for i in range(3):
...     print i
```

Boucle while

```
>>> i=0
>>> while i<3:
...     print i
...     i+=1
```

Structures conditionnelles et comparaisons

Condition if

```
for i in range(10):
   if i % 2 == 0:
      print str(i) + ' est pair'
   else:
      print str(i) + ' est impair'
```

break, continue

```
for n in range(2, 10):
   for x in range(2, n):
      if n % x == 0:
         print n, ' = ', x, '*', n/x
         break
      else:
         print  ' %s est un nombre premier' % n
```

Autres techniques pour les boucles

Pour les tableaux de très grande taille préférez xrange à range

```
$ python -m timeit 'for i in range(1000000):' ' pass'
10 loops, best of 3: 46.6 msec per loop
$ python -m timeit 'for i in xrange(1000000):' ' pass'
10 loops, best of 3: 32.2 msec per loop
```

Générateur enumerate :

```
>>> liste =  [1,2,3,5,7,11,13]
>>> for idx, ele in enumerate (liste):
...     print idx, ele
...
0 1
1 2
2 3
3 5
4 7
5 11
6 13
```

Les itérateurs

Un objet possédant une méthode **next()** qui renvoie à tour de rôle les valeurs pour chaque itération.

Exemple avec une liste

```
>>> liste = """Institut de Mecanique des Fluides
...                  et des Solides
...              """.split()
>>> iterateur = liste.__iter__()
>>> print iterateur.next()
Institut
>>> print iterateur.next()
de
>>> print iterateur.next()
Mecanique
```

Fonctions

Elle reçoit des arguments et renvoie éventuellement un résultat.
L'indentation délimite le corps de la fonction.

Fonction def

```
def norme(x, y):
    return sqrt(x*x+y*y)
>>> norme(3,4)
5
```

En l'absence de return une fonction renvoie la valeur None

Fonction lambda

```
norme = lambda x,y: sqrt(x*x+y*y)
>>> norme(3,4)
5
```

Ce type de fonction retourne une valeur unique

Gestion des paramètres et récursivité

Valeur par défaut et utilisation des paramètres par leurs noms :

```
>>> def somme(a,b=5):
...     return a+b
...
>>> somme(1)
6
>>> somme(b="a",a="bc")
'bca'
```

Fonction def avec un appel récursif

```
def fibo( n ):
    """ Retourne le nombre
         de Fibonacci n """
    if n == 0 or n == 1:
        return n
   else:
        return fibo( n - 1 ) + fibo( n - 2 )
print fibo(9)
help(fibo)
```

Déclaration d'une fonction sans connaître ses paramètres

Fonction à nombre variable de paramètres

```
>>> def somme(*args):
...     print args
...     sum(args)
...     return sum(args)
>>> add(4,5)
(4, 5)
9
```

Fonction à nombre variable de paramètres nommés

```
>>> def myfunc(**kwargs):
...     print kwargs
...
>>> myfunc(a=4, b='python', pi=3.14)
{'a': 4, 'pi': 3.14, 'b': 'python'}
```

Portée des variables

Chaque fonction possède son propre environnement.

```
>>> pi = 1.
>>> def deg2rad(theta):
...     pi = 3.14
...     return theta * pi / 180.
...
>>> deg2rad(45)
0.785
>>> pi
1.0
>>> def rad2deg(theta):
...     return theta*180./pi
...
>>> rad2deg(0.785)
141.3
>>> pi = 3.14
>>> rad2deg(0.785)
45.0
```

```
>>> def deg2rad(theta):
...     global pi
...     pi = 3.14
...     return theta * pi / 180
...
>>> pi = 1
>>> deg2rad(45)
0.785
>>> pi
3.14
>>>
```

Dans les fonctions, les variables ont une portée "locale". La déclaration en portée globale est déconseillée.

Les fonctions `map` et `zip`

- **zip** : permet de parcourir plusieurs séquences en parallèle
- **map** : applique une méthode sur une ou plusieurs séquences

$\Rightarrow$ map peut être beaucoup plus rapide qu'une boucle for.

```
>>> L1 = [1, 2, 3]
>>> L2 = [4, 5, 6]
>>> for (x, y) in zip(L1, L2):
...     print x, y, '--', x + y
...
1 4 -- 5
2 5 -- 7
3 6 -- 9
>>> from math import factorial
>>> map(factorial,range(4))
[1, 1, 2, 6]
>>> def add(x,y):
...     return x+y
...
>>> map(add,L1,L2)
[5, 7, 9]
```

Affécter les valeurs d'une fonction aux éléments d'une liste

Avec un `for`

```
f = range(10)
for i in range(1,10):
   f[i] = fibo(i)
   print f
```

Par compréhension

```
>>> li = [1, 9, 8, 4]
>>> print [elem*2 for elem in li]
[2, 18, 16, 8]
>>> print [n*n for n in range(1,10)]
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Une classe Python

Définition

```
class MaClasse:
   def __init__(self, i, j):
      self.i = i; self.j = j
   def write(self):
      print 'MaClasse: i=',self.i,'j=',self.j
```

L'argument `self` est obligatoire pour la définition.

Utilisation

```
>>> obj1 = MaClasse(6,9)
>>> print obj1.i, obj1.j
6 9
>>> obj1.write()
MaClasse: i= 6 j= 9
```

Héritage

Définition de la sous classe

```
class MaSousClasse(MaClasse):
   def __init__(self, i, j, k):
      MaClasse.__init__(self,i,j)
      self.k = k
   def write(self):
      print 'MaSousClasse: i=',self.i,'j=',self.j, \
          'k=', self.k
```

Utilisation

```
>>> obj1 = MaClasse(6,9)
>>> obj1.write()
MaClasse: i= 6 j= 9
>>> obj2 = MaSousClasse(6,9,12)
>>> obj2.write()
MaSousClasse: i = 6  j= 9  k =  12
```

Une classe qui s'utilise comme une fonction

Définition

```
class F:
   def __init__(self, a=1, b=1, c=1):
      self.a=a; self.b=b; self.c=c
   def __call__(self, x, y):
      return self.a+self.b*x+self.c*y*y
```

Appel

```
>>> f = F(a=2,b=4)
>>> v = f(2,1) + f(1.2,0)
>>> print v
17.8
```

Pratique lorsque l'on souhaite séparer les paramètres d'une fonction (a,b,c) des variables indépendantes (x,y).

Surcharge des opérateurs

Une classe vecteur

```
class Vecteur:
   dimension = 2
   def __init__(self, x=0, y=0):
      self.x=x
      self.y=y
   def __eq__(self, vB):
      return (self.x==vB.x) and (self.y==vB.y)
   def __add__(self, vB):
      return Vecteur(self.x+vB.x,self.y+vB.y)
   def __sub__(self, vB):
      return Vecteur(self.x-vB.x,self.y-vB.y)
   def __mul__(self, c):
      if isinstance(c,Vecteur):
         return self.x*c.x+self.y*c.y
      else:
         return Vecteur(c*self.x,c*self.y)
```

Surcharge des opérateurs (suite)

Manipulation

```
>>> U = Vecteur(); U.x,U.y = 0,1
>>> V = Vecteur(); V.x,V.y = 1,0
>>> print U, V
<__main__.Vecteur instance at 0x2b8ee0> <__main__.Vecteur instance at 0x2b8f30>
>>> W = U+V
>>> print W
<__main__.Vecteur instance at 0x2b8f80>
>>> Vecteur.dimension=3
>>> print 'Dimension de U=' + str(U.dimension)
Dimension de U=3
>>> print 'Dimension de V=' + str(V.dimension)
Dimension de V=3
>>> U.dimension=4
>>> print 'Dimension de V=' + str(V.dimension)
Dimension de V=3
>>> Vecteur.dimension=5
>>> print 'Dimension de U=' + str(U.dimension)
Dimension de U=4
>>> print 'Dimension de V=' + str(V.dimension)
Dimension de V=5
```

Entrées sorties

input et raw_input

```
>>> a=input("a? ")
a? [1,2,3]
>>> a
[1, 2, 3]
>>> type(a)
<type 'list'>
>>> a = raw_input("a? ")
a? [1,2,3]
>>> a
'[1,2,3]'
>>> type(a)
<type 'str'>
```

Création d'un objet fichier

```
f=open("fichier.tex")
for i in f:
 print (i)
f.close()
```

```
$ date > fichier.txt
$ python
>>> f=open("fichier.txt")
>>> for i in f:
...     print i
...
Ven 10 jui 2011 15:07:34 CEST

>>> f.close()
```

Modules pickle et h5py $\rightarrow$ exercices.

Les méthodes de l'objet fichier

- f.open(...)
- f.close()
- f.read() : lit l'ensemble du fichier et le renvoie sous forme de chaîne.
- f.readline() : lit et renvoie une ligne du fichier de f, retour inclus.
- f.readlines() : lit et renvoie une liste de toutes les lignes, où chaque ligne est représentée par une chaîne
- f.write(s) : écrit la chaîne s
- f.writelines(lst) : écrit la liste de chaîne lst

Module

dans un fichier fibo.py

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

Appel de la fonction fib

Méthode 1

```
import fibo
fibo.fib(1000)
```

Méthode 2

```
from fibo import fib
fib(1000)
```

Exécution d'un module python

```
def print_fib(n):
    "écrit la série de Fibonacci jusqu'à n"
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b
def list_fib(n):
    "retourne la série de Fibonacci jusqu'à n "
    result,a,b =[],0,1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
if __name__ == '__main__':
    print_fib(1000)
    print list_fib(100)
```

```
$ python fibo.py
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Importation d'un module python

- `import fibo`
- `import fibo as f`
- `from fibo import print_fib, list_fib`
- `from fibo import *` (importe tous les noms sauf variables et fonctions privées)

Remarque : En Python, les variables ou les fonctions privées commencent par _.
On peut accéder aux attributs du module avec :

- __dict__ : dictionnaire utilisé par le module pour l'espace de noms des attributs
- __name__ : nom du module
- __file__ : fichier du module
- __doc__ : documentation du module

Chemin de recherche d'un module python

Recherche dans sys.path

- dans le répertoire courant
- dans PYTHONPATH si défini (même syntaxe que PATH)
- dans un répertoire par défaut (sous Linux : /usr/lib/python)
- `from fibo import *` (importe tous les noms sauf variables et fonctions privées)

Ajout de mon dans sys.path

```
import sys
sys.path.append('le/chemin/de/mon/module')
import mon_module
```

Exemple d'un module avec différents répertoires

```
monModule/ Paquetage de niveau supérieur
__init__.py Initialisation du paquetage monModule
sous_module1/ Sous-paquetage
__init__.py
fichier1_1.py
fichier1_2.py ...
sous_module2/ Sous-paquetage
__init__.py
fichier2_1.py
fichier2_2.py ...
```

Le fichier __init__.py est obligatoire pour que Python considère les répertoires comme contenant des paquetages. Il peut-être vide ou contenir du code d'initialisation.

Les modules standards

- `sys` : variables système et accès aux options passées en ligne de commande.
- `os` : informations sur l'OS, manipulations de fichiers et gestions des processus.
  - getcwd() : renvoie le chemin menant au répertoire courant
  - abspath(path) : renvoie le chemin absolu de path
  - exists(path) : renvoie True si path désigne un fichier ou un répertoire existant, False sinon
- `math` : pi, sqrt, cos, sin, tan,...
- `string`
- `time`

Références

- La documentation officielle : `http://docs.python.org/tutorial`
- "Plongez au coeur de Python" : `http://diveintopython.adrahon.org`
- "Learning Python" de Mark Lutz et David Ascher chez O'Reilly
- "Python Scripting for Computational Science" de Hans Petter Langtangen chez Springer
- "How to Think Like a Computer Scientist : Learning with Python 2" `http://openbookproject.net//thinkCSpy/`
- "Cours de Python" de Patrick Fuchs et Pierre Poulain : `http://www.dsimb.inserm.fr/~fuchs/python/`
- "Calcul mathématique avec Sage" Paul Zimmermann et al : `http://sagebook.gforge.inria.fr/`.
- Formation Python en calcul scientifique `http://calcul.math.cnrs.fr/spip.php?article164`