Université Paris-Sud UE - M318
L3-MINT

Introduction à la programmation en langage Python

1. Présentation

Le Langage Python

Python est un langage de programmation (au même titre que le C, C++, fortran, java ...), développé en 1989. Ses principales caractéristiques sont les suivantes :
— «**open-source**» : son utilisation est gratuite et les fichiers sources sont disponibles et modifiables ;
— **simple et très lisible** ;
— doté d'une **bibliothèque de base très fournie** ;
— importante quantité de **bibliothèques disponibles** : pour le calcul scientifique, les statistiques, les bases de données, la visualisation ... ;
— **grande portabilité** : indépendant vis à vis du système d'exploitation (linux, windows, MacOS) ;
— **orienté objet** ;
— **typage dynamique** : le typage (association à une variable de son type et allocation zone mémoire en conséquence) est fait automatiquement lors de l'exécution du programme, ce qui permet une grande flexibilité et rapidité de programmation, mais qui se paye par une surconsommation de mémoire et une perte de performance ;
— présente un **support pour l'intégration d'autres langages**.

Comment faire fonctionner le code source ?

Il existe deux techniques principales pour traduire un code source en langage machine :
— la compilation : une application tierce, appelée compilateur, transforme les lignes de code en un fichier exécutable en langage machine. A chaque fois que l'on apporte une modification au programme, il faut recompiler avant de voir le résultat.
— l'interprétation : un interpréteur s'occupe de traduire ligne par ligne le programme en langage machine. Ce type de langage offre une plus grande commodité pour le développement, mais les exécutions sont souvent plus lentes.
Dans le cas de Python, on peut admettre pour commencer qu'il s'agit d'un langage interprété, qui fait appel à des modules compilés. Pour les opérations algorithmiques coûteuses, le langage Python peut s'interfacer à des bibliothèques écrites en langage de bas niveau comme le langage C.

Les différentes versions

Il existe deux versions de Python installées sur les machines enseignement du département de maths : 2.7 et 3.5 . Il faut maintenant privilégier la version 3.5 ; vérifiez que c'est bien celle-ci que vous utilisez.

L'interpréteur

Dans un terminal, taper `python` (interpréteur classique) ou `ipython` (interpréteur plus évolué) pour accéder à un interpréteur Python. L'écran affiche d'abord la version de `python` (normalement

3.X avec X entre 0 et 6, sinon réessayer avec `ipython3`) puis une aide minimaliste. Il propose ensuite à l'utilisateur d'entrer ces propres instructions (c'est l'invite de commande précédée de In). Vous pouvez maintenant taper dans ce terminal des instructions Python qui seront exécutées.

Le mode programmation

Il s'agit d'écrire dans un fichier une succession d'instructions qui ne seront effectuées que lorsque vous lancerez l'exécution du programme. Cela permet tout d'abord de sauvegarder les commandes qui pourront être réutilisées ultérieurement, et d'autre part d'organiser un programme, sous-forme de fichier principal, modules, fonctions ....

Le fichier à exécuter devra avoir l'extension `.py`, et devra contenir en première ligne le chemin pour accéder au compilateur Python, ainsi que l'encodage :

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
```

Il pourra être exécuté en lançant dans un terminal la commande `python nomdufichier.py`.

Utilisations de Python et librairies

— web : *Django, Zope, Plone,...*
— bases de données : *MySQL, Oracle,...*
— réseaux : *TwistedMatrix, PyRO, VTK,...*
— représentation graphique : *matplotlib, VTK,...*
— calcul scientifique : *numpy, scipy, ...*

Python pour le calcul scientifique

Python est une très bonne alternative aux logiciels de calcul scientifique classiques tels que Matlab, Octave, Scilab. D'une part c'est gratuit, d'autre part il s'agit d'un langage de programmation facile à apprendre et qui permet de faire des test rapides (langage interprété). Enfin, de nombreux modules sont disponibles pour le calcul scientifique.

L'environnement de programmation Spyder (sous Anaconda)

L'environnement Spyder permet de réaliser des programmes informatiques écrits avec le langage Python. Il est disponible avec la distribution Anaconda, qui présente de nombreux avantages, notamment celui d'être simple à installer. Son téléchargement se fait à l'adresse suivante : https ://store.continuum.io/cshop/anaconda/
Une fois la distribution Anaconda téléchargée et installée, on peut commencer à lancer Spyder on tapant `Spyder` dans un terminal.

Attention, sur les machines du SIF, lancer Spyder (version 3) on tapant `Spyder3` dans le terminal.

Au lancement de Spyder, apparaît une fenêtre partagée en deux zones. La zone en-bas à droite, appelée console (shell en anglais), est celle où l'on peut travailler de façon interactive avec l'interpréteur Python. La zone à gauche est un éditeur de texte, spécialement conçu pour écrire des programmes dans le langage Python.

2. Types et opérations de base

Le langage Python est orienté objet, c'est-à-dire qu'il permet de créer des objets, en définissant des attributs et des fonctions qui leur sont propres. Cependant, certains objets sont pré-définis dans le langage. Nous allons voir à présent les plus importants.

(a) **les nombres et les booléens**

- **entiers** (32 bits)
  type : `int`
- **réels** (64 bits)
  type : `float`
  exemples de valeurs : `4.  5.1  1.23e-6`
- **complexes**
  type : `complex`
  exemples de valeurs : `3+4j  3+4J`
- **booléens** type : `bool`
  exemples de valeurs : `True  False`

(b) **opérations de base**

- **affectation**

```
>>> i = 3    # i vaut 3
>>> a, k=True,  3.14159
>>> k=r=2.15
>>> x=complex(3,4)
```

- **affichage**

```
>>> i
3
>>> print(i)
3
```

- **Opérateurs addition, soustraction, multiplication, division**

```
 +, -,  *, \, \%,
```

- **Opérateurs puissance, valeur absolue**

```
**, pow, abs, \dots
```

- **Opérateurs de comparaison**

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

- **Opérateurs logiques**

```
or, and, not
```

- **Conversion**

```
>>> int(3.1415)
3
>>> float(3)
3.
```

(c) **les chaînes de caractères**

Une chaîne de caractères (string en anglais) est un objet de la classe (ou de type) `str`. Une chaîne de caractères peut être définie de plusieurs façons :

```
>>> "je suis une chaine"
```

```
'je suis une chaine'
```

```
>>> 'je suis une chaine'
'je suis une chaine'
```

```
>>> 'pour prendre l\'apostrophe'
'pour prendre l' apostrophe'
```

```
>>> "pour prendre l'apostrophe"
"pour prendre l'apostrophe"
```

```
>>> " " "ecrire
sur
plusieurs
lignes" " "
'ecrire\nsur\nplusieurs\nlignes'
```

— **concaténation**
On peut mettre plusieurs chaînes de caractères boût à boût.

```
>>> s = 'i vaut'
>>> i = 1
>>> print(s, i)
s vaut 1

>>> print(s+i)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects

>>> print(s+str(i))
>>> i vaut 1


>>> print ('*-'*5)
*-*-*-*-*-

>>> 'Hello '+"world!"
'Hello world!'
>>> x = 'Hello '
>>> y = "world!"
>>> z = x + y
>>> z
'Hello world!'
```

— **accès aux caractères**
Les caractères qui composent une chaîne sont numérotés à partir de zéro. On peut y accéder individuellement en faisant suivre le nom de la chaîne dÕun entier encadré par une paire de crochets :

```
>>> "bonjour"[3]; "bonjour"[-1]
'j'
'r'
>>> "bonjour"[2:]; "bonjour"[:3]; "bonjour"[3:5]
'njour'
'bon'
'jo'
```

```
>>> "bonjour"[-1::-1];
'ruojnob'
```

— **méthodes propres**
  — **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,
  — **...**

(d) **les listes**

Une liste consiste en une succession ordonnée d'objets, qui ne doivent pas nécessairement être du même type. Les termes d'une liste sont numérotés à partir de 0 (comme pour les chaînes de caractères).

— **initialisation**

```
>>> []; list();
[]
[]
>>> [1,2,3,4,5]; ['point','triangle','quad'];
[1, 2, 3, 4, 5]
['point', 'triangle', 'quad']
>>> [1,4,'mesh',4,'triangle',['point',6]];
[1, 4, 'mesh', 4, 'triangle', ['point', 6]]
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(2,10,2)
[2, 4, 6, 8]
```

— **modification**

Contrairement aux chaînes de caractères, on peut modifier les éléments d'une liste :

```
>>> l=[1,2,3,4,5]
>>> l[2:]=[2,2,2]
>>> l
[1, 2, 2, 2, 2]
```

— **concaténation**

```
>>> [0]*7
[0, 0, 0, 0, 0, 0, 0]
>>> L1, L2 = [1,2,3], [4,5]
>>> L1
[1, 2, 3]
>>> L2
[4, 5]
>>> L1+L2
[1, 2, 3, 4, 5]
```

— **méthodes propres**
  — **len(L)** : renvoie la taille de la liste L,
  — **L.sort** : trie la liste L,
  — **L.append** : ajoute un élément à la fin de la liste L,
  — **L.reverse** : inverse la liste L,
  — **L.index** : recherche un élément dans la liste L,
  — **L.remove** : retire un élément de la liste L,
  — **L.pop** : retire le dernier élément de la liste L,

— ...

(e) **copie d'un objet**

```
>>> L = ['Dans','python','tout','est','objet']
>>> T = L
>>> T[4] = 'bon'
>>> T
['Dans', 'python', 'tout', 'est', 'bon']
>>> L
['Dans', 'python', 'tout', 'est', 'bon']
>>> L=T[:]
>>> L[4]='objet'
>>> T;L
['Dans', 'python', 'tout', 'est', 'bon']
['Dans', 'python', 'tout', 'est', 'objet']
```

(f) **quelques remarques importantes**

— en Python, tout est objet,
— une chaîne de caractère est immuable, tandis qu'une liste est muable,
— **type** permet de connaître le type d'un objet,
— **id** permet de connaître l'adresse d'un objet,
— **eval** permet d'évaluer une chaîne de caractères.

3. **Commentaires**

```
# ceci est un commentaire
```

4. **Noms de variables**

Python fait la distinction entre minuscules et majuscules.
Conseil : donner des noms significatifs aux variables et aux fonctions.

5. **Les structures de contrôle**

(a) **L'indentation**

Les fonctions Python n'ont pas de `begin` ou `end` explicites, ni d'accolades qui pourraient marquer là ou commence et ou se termine le code de la fonction. Le seul délimiteur est les deux points et l'indentation du code lui-même. Les blocs de code (fonctions, instructions if, boucles for ou while etc) sont définis par leur indentation. L'indentation démarre le bloc et la désindendation le termine. Il n'y a pas d'accolades, de crochets ou de mots clés spécifiques. Cela signifie que les espaces blancs sont significatifs et qu'ils doivent être cohérents. Voici un exemple :

```
a = -150
if a < 0:
    print('a est negatif')

ligne d'en-tête
    première instruction du bloc
    . . .
    dernière instruction du bloc
```

Fonctionnement par blocs :

```
Bloc 1
 ...
Ligne d'en-tête :
      Bloc 2
       ...
      Ligne d'en-tête :
            Bloc 3
             ...
            Ligne d'en-tête :
      Bloc 2 (suite)
       ...
Bloc 1 (suite)
 ...
```

(b) **Le test de conditions**

Le test de condition se fait sous la forme générale suivante :

```
if < test1 > :
      < blocs d'instructions 1>
elif < test2 > :
      < blocs d'instructions 2 >
else :
      < blocs d'instructions 3 >
```

Un exemple :

```
a = 10.
if a > 0:
      print("a est strictement positif")
      if a >= 10:
            print (' a est un nombre ')
      else:
            print (' a est un chiffre ')
            a += 1
elif a is not 0:
      print(' a est strictement negatif ')
else:
      print(' a est nul ')
```

Un autre exemple :

```
L = [1, 3, 6, 8]
if 9 in L:
      print('9 est dans la liste L')
else:
      L.append(9)
```

(c) **La boucle conditionnelle**

La forme générale d'une boucle conditionnelle est

```
while < test1 > :
      < blocs d'instructions 1 >
      if < test2 > : break
      if < test3 > : continue
else :
      < blocs d'instructions 2 >
```

où l'on a utilisé les méthodes suivantes :

break : sort de la boucle sans passer par else,
continue : remonte au début de la boucle,
pass : ne fait rien.
La structure finale **else** est optionnelle. Elle signifie que l'instruction est lancée si et seulement si la boucle se termine normalement.

Quelques exemples :

— Boucle infinie :
```
while 1:
      pass
```
— y est-il premier ?
```
x = y/2
while x > 1 :
      if y%x ==0
            print (str(y)+' est facteur de '+str(x))
            break
      x = x - 1
else :
     print( str(y)+ ' est premier')
```

(d) **La boucle inconditionnelle**

La forme générale d'une boucle inconditionnelle est

```
for < cible > in < objet > :
      < blocs d'instructions 1 >
      if < test1 > : break
      if < test2 > : continue
else :
      < blocs d'instructions 2 >
```

Quelques exemples :

```
sum = 0
for i in [1, 2, 3, 4] :
      sum += 1

prod = 1
for p in range(1, 10) :
      prod *= p

s = 'bonjour'
for c in s :
      print c,

L = [ x + 10 for x in range(10) ]
```

Si l'on veut itérer sur un grand nombre d'éléments, il vaut mieux utiliser l'itérateur `range(start,stop,step)` qui ne passe pas par le construction d'une liste ou d'un tableau et permet ainsi un calcul légèrement plus rapide et une économie en taille mémoire.

6. **Les fonctions**

La structure générale de définition d'une fonction est la suivante

```
def < nom fonction > (arg1, arg2,... argN):
          ...
          bloc d'instructions
          ...
          return < valeur( s ) >
```

Voici quelques exemples :

```
def table7():
          n=1
          while n < 11:
          print n*7
          n+=1
```

Un fonction qui n'a pas de `return` renvoie par défaut `None`.

```
def table(base):
          n=1
          while n < 11:
          print n*base
          n+=1
```

```
def table(base, debut=0, fin=11):
          print ('Fragment de la table de multiplication par '+str(base)+Õ : Õ)
          n=debut
          l=[]
          while n < fin:
          print n*base
          l.append(n*base)
          n+=1
          return l
```

On peut également déclarer une fonction sans connaître ses paramètres :

```
>>> def f (*args, **kwargs):
          ... print(args)
          ... print(kwargs)

>>> f(1,3,'b',j=1)
(1, 3, 'b')
{'j': 1}
```

Il existe une autre façon de déclarer une fonction de plusieurs paramètres :

```
lambda argument 1, ... , argument N : expression utilisant les arguments
```

Exemple :

```
>>> f = lambda x, i :  x**i
>>> f(2,4)
16
```

7. Les modules

Un module est un fichier comprenant un ensemble de définitions et dÕinstructions compréhensibles par Python. Il permet d'étendre les fonctionnalités du langage. Voici tout d'abord un exemple de module permettant l'utilisation des nombres de Fibonacci.

Fichier fibo.py

```
# Module nombres de Fibonacci
def print_fib(n) :
            " " "
            ecrit la serie de Fibonacci jusquÕa n
            " " "
            a, b = 0, 1
            while b < n:
                  print(b),
                  a, b = b, a + b
            print

def print_fib(n) :
            " " "
            retourne la serie de Fibonacci jusquÕa n
            " " "
            result, a, b = [], 0, 1
            while b < n:
                  result.append(b),
                  a, b = b, a + b
            return result
```

Utilisation du module fibo.py

```
>>> import fibo
>>> fibo.print_fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.list_fib(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

L'importation de modules

Il existe plusieurs façons d'importer un module :

— **import** fibo
— **import** fibo **as** f
— **from** fibo **import** print_ fib, list_ fib
— **from** fibo **import** * (importe tous les noms sauf les variables et les fonctions privées)

Remarques :
En Python, les variables ou les fonctions privées commencent par _.
Lors de l'exécution d'un programme le module est importé une seule fois.
Pour le recharger : reload(M) si utilisation de import M.
Attention : from M import A

reload(M) n'aura aucune incidence sur l'attribut A du module M.

Chemin de recherche d'un module

Recherche dans sys.path : dans le répertoire courant, dans PYTHONPATH si défini, dans un répertoire par défaut (sous LINUX : /usr/lib/python).

Ajout d'un module dans sys.path :

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

Exemple de module avec différents paquetages

```
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, mais il peut-être vide. Il peut contenir du code d'initialisation et/ou la variable _all_.

Exemple de fichier `monModule/sous_module2/_init_.py` :

```
_all_ = ["fichier2_1", "fichier2_2"]
```

Utilisation :

```
>>> from monModule.sous_module2 import *
```

Importe les attributs et fonctions se trouvant dans fichier2_1 et fichier2_2. On y accéde en tapant fichier2_1.mon_attribut.

Le module math

Ce module fournit un ensemble de fonctions mathématiques pour les réels :

```
pi
sqrt
cos, sin, tan, acos, ...
...
```

Tapez `import math` pour importer le module, puis `help(math)` pour avoir toutes les informations sur le module : chemin, fonctions, constantes.