Introduction au langage Python pour le calcul scientifique

Loïc Gouarin

Laboratoire de Mathématiques d'Orsay

23 mai 2013

Plan

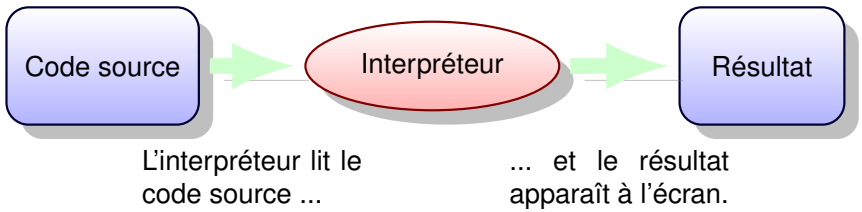
Le langage Python

1. développé en 1989 par Guido van Rossum
2. open-source
3. portable
4. orienté objet
5. dynamique
6. extensible
7. support pour l'intégration d'autres langages

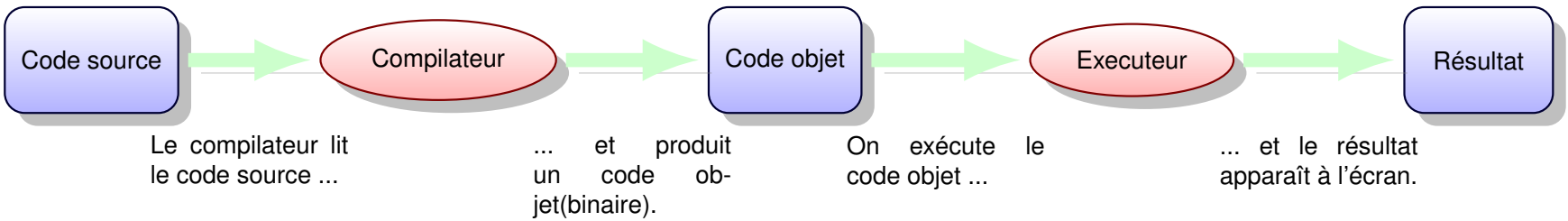
Comment faire fonctionner mon code source ?

Il existe 2 techniques principales pour effectuer la traduction en langage machine de mon code source :

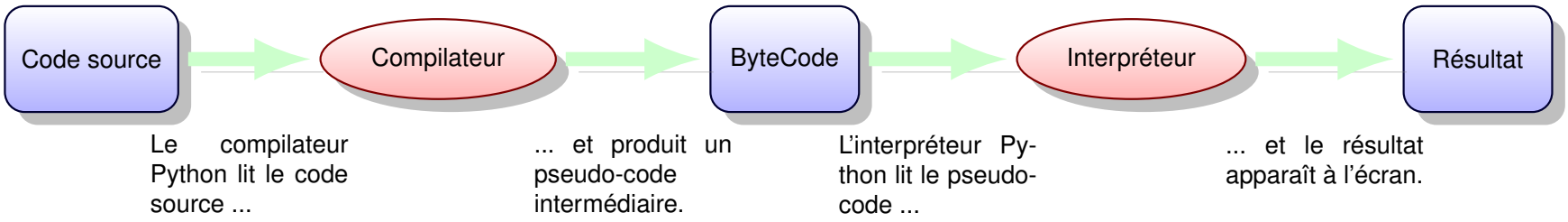
- Interprétation



- Compilation



Et Python ?



Avantages :

- interpréteur permettant de tester n'importe quel petit bout de code,
- compilation transparente.

Inconvénients :

- peut être lent.

Les différentes implémentations

- **CPython**
  Implémentation de base basé sur le langage C ANSI
- **Jython**
  Implémentation permettant de mixer Python et java dans la même JVM
- **IronPython**
  Implémentation permettant d'utiliser Python pour Microsoft .NET
- **PyPy**
  Implémentation de Python en Python
- **...**

Les différentes versions

- Il existe 2 versions de Python : 2.7 et 3.3.
- 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.

L'interpréteur

Sous Linux

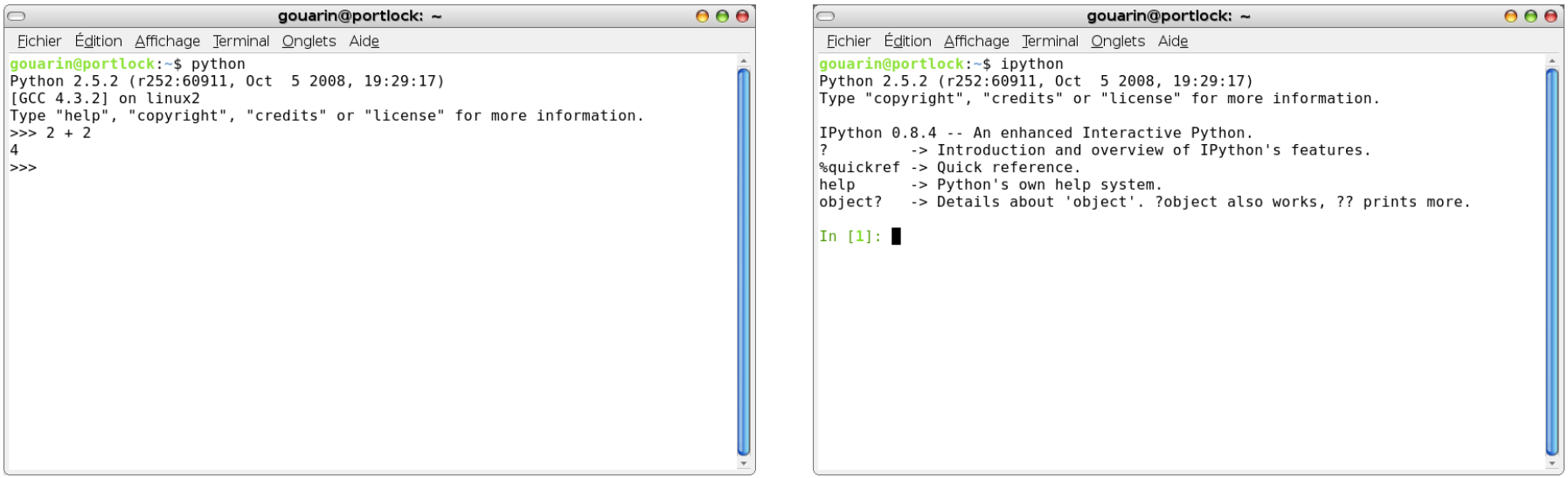


FIGURE: Interpréteur classique (gauche) et ipython (droite)

Options utiles de l'interpréteur

- `-c` : exécute la commande Python entrée après,
- `-i` : passe en mode intéractif après avoir exécuter un script ou une commande,
- `-d` : passe en mode debug.

Que peut-on faire avec Python ?

- **web**
  Django, TurboGears, Zope, Plone, ...
- **bases de données**
  MySQL, PostgrSQL, Oracle, ...
- **réseaux**
  TwistedMatrix, PyRO, ...
- **Gui**
  Gtk, Qt, Tcl/Tk, WxWidgets
- **représentation graphique**
  gnuplot, matplotlib, VTK, ...
- **calcul scientifique**
  numpy, scipy, sage, ...
- **...**

Pourquoi utiliser Python pour le calcul scientifique ?

- peut être appris en quelques jours
- permet de faire des tests rapides
- alternative à Matlab, Octave, Scilab, ...
- parallélisation
- tourne sur la plupart des plateformes
- nombreux modules pour le calcul scientifique

entiers (32 bits) :

```
0    -13    124
```

entiers longs (précision illimitée) :

```
1L    340282366920938463463374607431768211456
```

réels (64 bits) :

```
5.    1.3    -4.7    1.23e-6
```

complexes :

```
3 + 4j, 3 + 4J
```

booléens :

```
True    False
```

Opérations de base

commentaire

```
# ceci est un commentaire
```

nom de variables

En Python, les majuscules et les minuscules dans les noms de variables sont différenciées.

```
pi, Pi, PI # 3 variables differentes
```

Opérations de base

affectation

```
>>> i = 3 # i vaut 3
>>> a, pi = True, 3.14159
>>> k = r = 2.15
```

affichage dans l'interpréteur

```
>>> i
3
>>> print i
3
```

Opérations de base

Opérateurs addition, soustraction, multiplication et division

```
+, -, *, /, %, //
```

Opérateurs puissance, valeur absolue, ...

```
**, pow, abs, ...
```

Opérateurs de comparaisons

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

Opérateurs bitwise

```
&, ^ , |, <<, >>
```

Opérateurs logiques

```
or, and, not
```

Opérations de base

conversion

```
>>> int(3.1415) # conversion d'un float en int
3
>>> float(3) # conversion d'un int en float
3.0
>>> long(5) # conversion d'un int en long int
5L
```

Règles de priorité

PEMDAS

1. Parenthèses : `2*(3 - 1) = 4`
2. Exposant : `4**2 - 1 = 15`
3. Multiplication/Division : `-1 + 3*5 + 2 = 16`
4. Addition/soustraction

Division avec les entiers

```
>>> 3/4
0
>>> 3%4
3
>>> 3./4
0.75
```

Manipulations de chaînes de caractères

Définir une chaîne

```
>>> "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 ...'
```

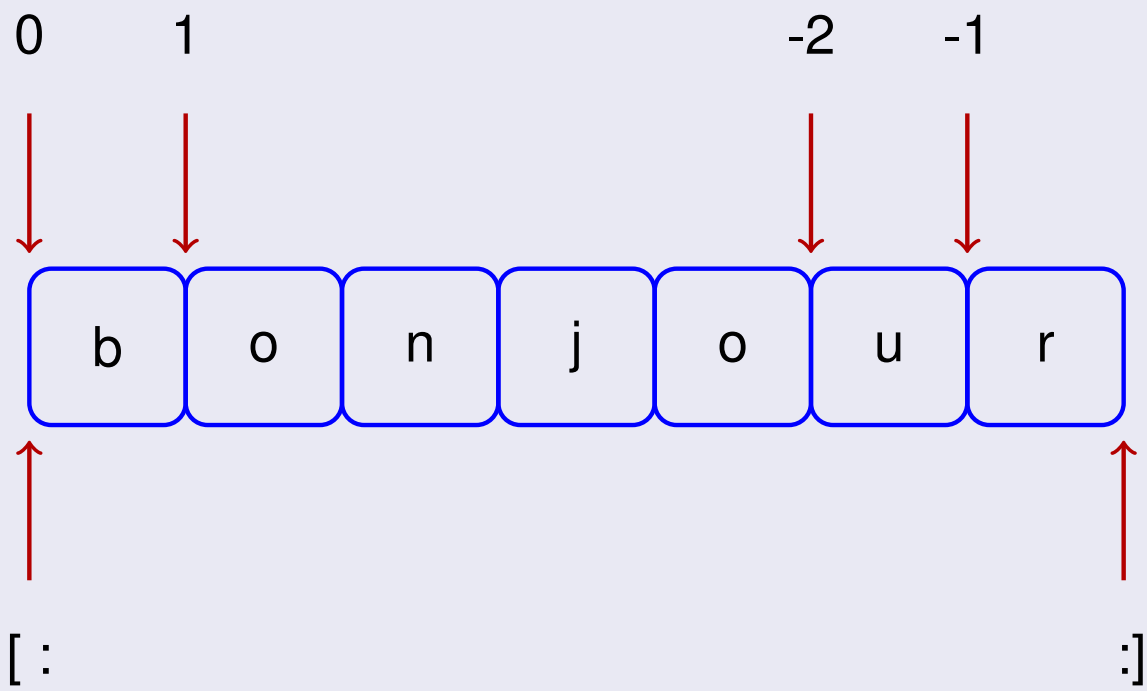
Manipulations de chaînes de caractères

Concaténation

```
>>> s = 'i vaut'
>>> i = 1
>>> print s + i
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>> print s + " %d %s"%(i, "m.")
i vaut 1 m.
>>> print s + ' ' + str(i)
i vaut 1
>>> print '*-'*5
*-*-*-*-*-
```

Manipulations de chaînes de caractères

Accès aux caractères [debut : fin : pas]



Manipulations de chaînes de caractères

Accès aux caractères

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

Une chaîne est un objet immutable.

Une chaîne s a ses propres méthodes (`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 contient que des nombres, False sinon,
- ...

Petit aparté

- en python, tout est objet
- **dir** permet de voir les objets et méthodes disponibles
- **help** permet d'avoir une aide
- **type** permet de connaître le type de l'objet
- **id** permet d'avoir l'adresse d'un objet
- **eval** permet d'évaluer une chaîne de caractères
- **input** et **raw_input** sont l'équivalent du *scanf* en C

Petit aparté (suite)

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
```

Initialisation

```
[]; list();
[1, 2, 3, 4, 5]; ['point', 'triangle', 'quad'];
[1, 4, 'mesh', 4, 'triangle', ['point', 6]];
range(10)}; range(2, 10, 2)
```

Concaténation

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

Une liste est une séquence comme pour les chaînes de caractères.

Copie d'une liste

ATTENTION !

```
>>> 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']
```

Une liste L a ses propres méthodes (`help(list)`)

- **len(L)** : taille de la liste
- **L.sort** : trier la liste L
- **L.append** : ajout d'un élément à la fin de la liste L
- **L.reverse** : inverser la liste L
- **L.index** : rechercher un élément dans la liste L
- **L.remove** : retirer un élement de la liste L
- **L.pop** : retirer le dernier élément de la liste L
- ...

Initialisation

```
(); tuple();
(1,); 'a', 'b', 'c', 'd',
('a', 'b', 'c', 'd')
```

Concaténation

```
>>> (1, 2)*3
(1, 2, 1, 2, 1, 2)
>>> t1, t2 = (1, 2, 3), (4, 5)
>>> t1 + t2
(1, 2, 3, 4, 5)
```

Un tuple est aussi une séquence.

Opérations sur un tuple

un tuple n'est pas modifiable

```
>>> t = 'a', 'b', 'c', 'd'
>>> t[0] = 'alpha'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support item assignment
>>> t= ('alpha',) + t[1:]
>>> t
('alpha', 'b', 'c', 'd')
```

mais un objet modifiable dans un tuple peut l'être

```
>>> t = (1, 2, [3, 4], 6)
>>> t[2][0] = 1; t
(1, 2, [1, 4], 6)
```

Initialisation

```
{}; dict(); {'point': 1, 'ligne': 2, 'triangle': 3}
```

Remarques

- un dictionnaire n'est pas une séquence
- un dictionnaire est constitué de clés et de valeurs
- on ne peut pas concaténer un dictionnaire avec un autre

Ajout d'une clé ou modification d'une valeur

```
>>> dico['quad'] = 4
>>> dico
{'quad': 4, 'ligne': 2, 'triangle': 3, 'point': 1}
>>> dico['point'] = 3
{'quad': 4, 'ligne': 2, 'triangle': 3, 'point': 3}
```

Copie d'un dictionnaire

```
>>> dico = {'computer':'ordinateur', 'mouse':'souris',
... 'keyboard':'clavier'}
>>> dico2 = dico
>>> dico3 = dico.copy()
>>> dico2['printer'] = 'imprimante'
>>> dico2
{'computer': 'ordinateur', 'mouse': 'souris',
 'printer': 'imprimante', 'keyboard': 'clavier'}
>>> dico
{'computer': 'ordinateur', 'mouse': 'souris',
 'printer': 'imprimante', 'keyboard': 'clavier'}
>>> dico3
{'computer': 'ordinateur', 'mouse': 'souris',
 'keyboard': 'clavier'}
```

Un dictionnaire a ses propres méthodes (`help(dict)`)

- **len(dico)** : taille du dictionnaire
- **dico.keys** : renvoie les clés du dictionnaire sous forme de liste
- **dico.values** : renvoie les valeurs du dictionnaire sous forme de liste
- **dico.has_key** : renvoie True si la clé existe, False sinon
- **dico.get** : donne la valeur de la clé si elle existe, sinon une valeur par défaut
- ...

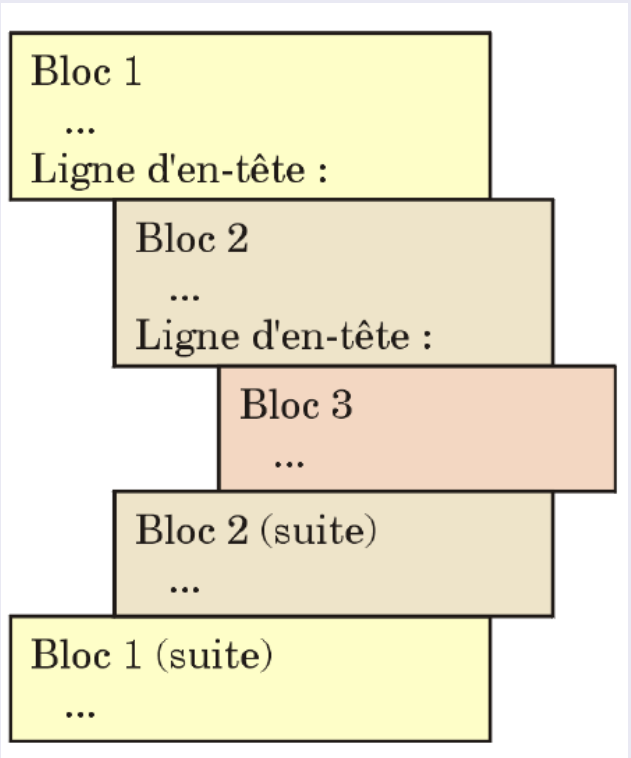
Un petit exemple

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

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

Indentation générale

Fonctionnement par blocs



Code sur plusieurs lignes

Cas1

```
>>> a = 2 + \
... 3*2
```

Cas2

```
>>> l = [1,
...      2]
>>> d = { 1:1,
...        2:2}
>>> b = 2*(5 +
...        5*2)
```

Format général

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

Exemple 1

```
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'
```

Exemple 2

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

Format général

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

- **break** : sort de la boucle sans passer par else,
- **continue** : remonte au début de la boucle,
- **pass** : ne fait rien,
- **else** : lancé si et seulement si la boucle se termine normalement.

Exemples

boucle infinie

```
while 1:
    pass
```

y est-il premier ?

```
x = y/2
while x > 1:
    if y%x == 0:
        print y, 'est facteur de', x
        break
    x = x - 1
else:
    print y, 'est premier'
```

Format général

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

Exemples :

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

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)]
```

Remarque

Pour un grand nombre d'éléments, on préférera utiliser **xrange** plutôt que **range**.

Définition

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

Remarque

map peut être beaucoup plus rapide que l'utilisation de **for**

Exemples

Utilisation de zip

```
L1 = [1, 2, 3]
L2 = [4, 5, 6]

for (x, y) in zip(L1, L2):
    print x, y, '--', x + y
```

Utilisation de map

```
S = '0123456789'
print map(int, S)
```

Autre exemple

```
S1 = 'abc'
S2 = 'xyz123'

print zip(S1, S2)
print map(None, S1, S2)
```

Définition

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

Exemples

Fonction sans paramètres

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

Remarque

Une fonction qui n'a pas de **return** renvoie par défaut **None**.

Exemples

Fonction avec paramètre

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

Exemples

Fonction avec plusieurs paramètres

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

Déclaration d'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}
```

lambda

Définition

```
lambda argument1,... argumentN : expression utilisant les arguments
```

Exemple

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

Création d'un objet fichier avec open

f = open(filename, mode = 'r', bufsize = -1)

- **'r'** : le fichier, qui doit déjà exister, est ouvert en lecture seule.
- **'w'** : le fichier est ouvert en écriture seule. S'il existe déjà, il est écrasé ; il est créé sinon.
- **'a'** : le fichier est ouvert en écriture seule. Son contenu est conservé.
- l'option **'+'** : le fichier est ouvert en lecture et en écriture.
- l'option **'b'** : ouverture d'un fichier binaire.

Attributs et méthodes des objets fichiers

- **f.close()** : ferme le fichier
- **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**, la fin de ligne ($\backslash n$) incluse.
- **f.readlines()** : lit et renvoie une liste de toutes les lignes du fichier de **f**, où chaque ligne est représentée par une chaîne se terminant par $\backslash n$
- **f.write(s)** : écrit la chaîne **s** dans le fichier de **f**
- **f.writelines(lst)** : écrit la liste de chaînes **lst** dans le fichier de **f**

Exemples

Exemple 1

```
# copie bete de 2 fichiers
fr = open('orig.txt')
fw = open('copy.txt', 'w')

lines = fr.read()
fw.write(lines)
fr.close()
fw.close()
```

Exemple 2

```
with open('donnees.txt') as f:
    # ...
```

Les classes

Définition

```
class <nom_classe>(superclass,...):
    donnee = valeur
    def methode(self,...):
        self.membre = valeur
```

Objet classe

admet 2 types d'opérations :

- référenciation des attributs
- instanciation

Référenciation des attributs

- peut être une variable, une fonction, ...
- syntaxe standard utilisée pour toutes les références d'attribut en Python : obj.nom
- valide si l'attribut fait partie de la classe

Exemple

```
class MaClasse:
    """
    Une classe simple pour exemple
    """
    i = 12345
    def f(self):
        return 'bonjour'
```

- **MaClasse.i** : référence d'attribut valide ; renvoie un entier
- **MaClasse.f** : référence d'attribut valide ; renvoie un objet fonction

Les classes

Instance

- utilise la notation d'appel de fonction
- renvoie une instance de la classe

Exemple

```
x = MaClasse()
```

Les classes

Initialisation

- dans le cas précédent, création d'un objet vide
- __init__ : fonction permettant d'initialiser la classe

Exemple

```
>>> class Complexe:
...     def __init__(self, reel, imag):
...         self.r = reel
...         self.i = imag
...
>>> x = Complexe(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

Les autres méthodes

```
class vecteur:
    def __init__(self, x, y, z = 0):
        self.coords = [x, y, z]

    def __str__(self):
        s = ''
        for c in self.coords:
            s += '( ' + str(c) + ' )\n'
        return s

    def __add__(self, v):
        return vecteur(self.coords[0] + v.coords[0],
                       self.coords[1] + v.coords[1],
                       self.coords[2] + v.coords[2])
```

Les autres méthodes

```
>>> v1 = vecteur(1, 2)
>>> v2 = vecteur(4.1, 3.4, 1.)
>>> v3 = v1 + v2
>>> print v3
( 5.1 )
( 5.4 )
( 1.0 )
```

Qu'est-ce qu'un module ?

Un module est un fichier comprenant un ensemble de définitions et d'instructions compréhensibles par Python.

Il permet d'étendre les fonctionnalités du langage.

Exemple : fibo.py

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

Exemple : fibo.py (suite)

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

Utilisation du module fibo

```
>>> 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

Les différentes manières 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 variables et fonctions privées)

Remarque : En Python, les variables ou les fonctions privées commencent par _.

L'importation

Compléments sur *import*

import définit explicitement certains attributs du module :

- **__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

L'importation

Remarques

- lors de l'exécution d'un programme le module est importé qu'une seule fois
- possibilité de 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

Exécution d'un module

Ajout à la fin de fibo.py

```
if __name__ == '__main__':
    print_fib(1000)
    print list_fib(100)
```

Résultat

```
$ 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]
```

Chemin de recherche d'un module

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)

Ajout de mon module dans sys.path

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

Recherche du fichier d'un module M

- .pyd et .dll (windows) ou .so (linux)
- .py
- .pyc
- dernier chemin : M/__init__.py

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

- Obligatoire pour que Python considère les répertoires comme contenant des paquetages
- peut-être vide
- peut contenir du code d'initialisation
- peut contenir la variable __**all**__

Le fichier __init__.py

Exemple monModule/sous_module2/__init__.py

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

Uitlisation

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

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

Les modules standards

- sys
- os
- re
- string
- math
- time
- ...

Présentation du module sys

- information système (version de python)
- options du système
- récupération des arguments passés en ligne de commande

sys.path

- donne le python path où sont recherchés les modules lors de l'utilisation d'import
- sys.path est une liste
  pour ajouter un élément : sys.path.append('...')
- le premier élément est le répertoire courant

sys.exit

sys.exit permet de quitter un script python.

sys.argv

sys.argv permet de récupérer les options passées en ligne de commandes.

Présentation du module os

- permet de travailler avec les différents systèmes d'exploitation
- création de fichiers, manipulation de fichiers et de répertoires
- création, gestion et destruction de processus

os.name

Chaîne de caractères définissant le type de plateforme sur laquelle s'exécute Python :

- posix : système unix + MacOs X
- nt : windows
- mac : mac avant MacOs X
- java : jython

Fonctions du module os sur les fichiers et les répertoires

- **getcwd()** : renvoie le chemin menant au répertoire courant
- **abspath(path)** : renvoie le chemin absolu de path
- **listdir(path)** : renvoie une liste contenant tous les fichiers et sous-répertoires de path
- **exists(path)** : renvoie True si path designe un fichier ou un répertoire existant, False sinon
- **isfile(path)** : renvoie True si path est un fichier, False sinon
- **isdir(path)** : renvoie True si path est un répertoire, False sinon
- **...**

Présentation du module math

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

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

Le module numpy

- outils performants pour la manipulation de tableaux à *N* dimensions
- fonctions basiques en algèbre linéaire
- fonctions basiques pour les transformées de Fourier
- outils pour intégrer du code Fortran
- outils pour intégrer du code C/C++
- outils d'installation avec support tests unitaires

Comparaison syntaxique numpy, Matlab, C

on veut calculer

$u = 100 \exp\left(-100(x - 0.5)^2\right)$ avec $x \in [0, 1]$.

en Python

```
from numpy import *
x = linspace(0., 1., 100)
u = 100.*exp(-100.*(x - .5)**2)
```

en Matlab

```
x = linspace(0.,1.,100)
u=100.*exp(-100.*(x-.5).^2)
```

Comparaison syntaxique numpy, Matlab, C

en C

```c
#include <stdio.h>
#include <math.h>
#define N 100

int main(void)
{
    int i;
    double dx=1./(N-1);
    double x=0., u[N];

    for (i=0;i<N;i++)
        {
            u[i] = 100.*exp(-100.*(x-.5)*(x-.5));
            x += dx;
         }
  return 0;
}
```

Création d'un tableau en connaissant sa taille

```
>>> import numpy as np
>>> a = np.zeros(4)
>>> a
array([ 0.,  0.,  0.,  0.])
>>> nx, ny = 2, 2
>>> a = np.zeros((nx, ny, 2))
>>> a
array([[[ 0.,  0.],
        [ 0.,  0.]],

       [[ 0.,  0.],
        [ 0.,  0.]]])
```

Il existe également

`np.ones`, `np.eye`, `np.identity`, `np.empty`, ...

Création d'un tableau en connaissant sa taille

- Par défaut les éléments sont des float (équivalent de double en C).
- On peut donner un deuxième argument qui précise le type (int, complex, bool, ...).
- Voir également les méthodes dans random.

Création d'un tableau avec une séquence de nombre

```
>>> a = np.linspace(-4, 4, 9)
>>> a
array([-4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.])
>>> a = np.arange(-4, 4, 1)
>>> a
array([-4, -3, -2, -1,  0,  1,  2,  3])
```

Création d'un tableau à partir d'une séquence

```
>>> a = np.array([1, 2, 3])
>>> a
array([1, 2, 3])
>>> b = np.array(range(10))
>>> b
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> L1, L2 = [1, 2, 3], [4, 5, 6]
>>> a = np.array([L1, L2])
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
```

Création d'un tableau à partir d'une fonction

```
>>> def f(x, y):
...     return x**2 + np.sin(y)
...
>>> a = np.fromfunction(f, (2, 3))
>>> a
array([[ 0.        ,  0.84147098,  0.90929743],
       [ 1.        ,  1.84147098,  1.90929743]])
```

Création d'un tableau d'objets

```
>>> class vecteur:
...     def __init__(self, x, y, z):
...         self.x, self.y, self.z = x, y, z
...
>>> np.array([vecteur(1,1,1), vecteur(2,1,1)])
array([<__main__.vecteur instance at 0x19db518>,
       <__main__.vecteur instance at 0x19dbb90>], dtype=object
```

Création d'un tableau avec des records

On crée une image contenant 2x2 pixels

```
>>> img = np.array([[[0, 1], [0, 0]],
...                 [[0, 0], [1, 0]],
...                 [[0, 0], [0, 1]]], dtype=np.float32)
>>> img
array([[[ 0.,  1.],
        [ 0.,  0.]],

       [[ 0.,  0.],
        [ 1.,  0.]],

       [[ 0.,  0.],
        [ 0.,  1.]]], dtype=float32)
```

Création d'un tableau avec des records

Pixel rouge

```
>>> img[0]
array([[ 0.,  1.],
       [ 0.,  0.]], dtype=float32)
```

Pixel en haut à droite

```
>>> img[:, 0, 1]
array([ 1.,  0.,  0.], dtype=float32)
```

Création d'un tableau avec des records

Avec des records

```
>>> img = np.array([[(0, 0, 0), (1, 0, 0)],
...                 [(0, 1, 0), (0, 0, 1)]],
...                 [('r', np.float32),
...                  ('g', np.float32),
...                  ('b', np.float32)])
>>> img['r']
array([[ 0.,  1.],
       [ 0.,  0.]], dtype=float32)
>>> img[0, 1]
(1.0, 0.0, 0.0)
```

Caractéristiques d'un tableau a

- **a.shape** : retourne les dimensions du tableau
- **a.dtype** : retourne le type des éléments du tableau
- **a.size** : retourne le nombre total d'éléments du tableau
- **a.ndim** : retourne la dimension du tableau

Indexation

```
>>> L1, L2 = [1, -2, 3], [-4, 5, 6]
>>> a = np.array([L1, L2])
>>> a[1, 2]
6
>>> a[:, 1]
array([-2, 5])
>>> a[:, -1:0:-1]
array([[3, -2],
       [6, 5]])
>>> a[a < 0]
array([-2, -4])
```

Copie d'un tableau

```
>>> a = np.linspace(1, 5, 5)
>>> b = a
>>> c = a.copy()
>>> d = np.zeros(a.shape, a.dtype)
>>> d[:] = a
>>> b[1] = 9
>>> a; b; c; d
array([ 1.,  9.,  3.,  4.,  5.])
array([ 1.,  9.,  3.,  4.,  5.])
array([ 1.,  2.,  3.,  4.,  5.])
array([ 1.,  2.,  3.,  4.,  5.])
```

Redimensionnement d'un tableau

```
>>> a = np.linspace(1, 10, 10)
>>> a.shape = (2, 5)
>>> a
array([[  1.,   2.,   3.,   4.,   5.],
       [  6.,   7.,   8.,   9.,  10.]])
>>> a.shape = (a.size,)
>>> a.reshape((2, 5))
array([[  1.,   2.,   3.,   4.,   5.],
       [  6.,   7.,   8.,   9.,  10.]])
```

Attention : **reshape** ne fait pas une copie du tableau mais crée une nouvelle vue.

Rangement mémoire

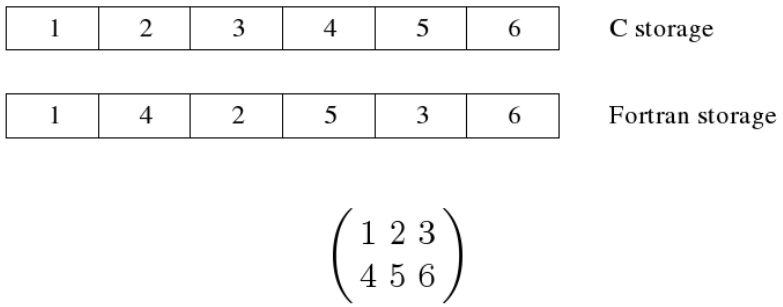


FIGURE: figure tirée de *Python Scripting for Computational Science*

```
>>> a = np.array([[1, 2], [3, 4]], order = 'f')
>>> b = np.array([[1, 2], [3, 4]])
>>> np.isfortran(a)
True
>>> np.isfortran(b)
False
```

Boucles sur les tableaux

```
>>> a = np.zeros((2, 3))
>>> for i in xrange(a.shape[0]):
...     for j in xrange(a.shape[1]):
...         a[i, j] = (i + 1)*(j + 1)
>>> print a
[[ 1.  2.  3.]
 [ 2.  4.  6.]]
>>> for e in a:
...     print e
[ 1.  2.  3.]
[ 2.  4.  6.]
```

Boucles sur les tableaux

```
>>> for e in a.flat:
...     print e
1.0
2.0
3.0
2.0
4.0
6.0
```

Boucles sur les tableaux

```
>>> for index, value in np.ndenumerate(a):
...     print index, value
(0, 0) 1.0
(0, 1) 2.0
(0, 2) 3.0
(1, 0) 2.0
(1, 1) 4.0
(1, 2) 6.0
```

Calculs sur les tableaux

- il n'est pas nécessaire de faire des boucles
- Python sait faire la différence entre un tableau numpy et un scalaire
- le calcul sur les tableaux se fait via des fonctions C

Calculs sur les tableaux

Exemple

On veut calculer b = 3a - 1 !

```
In [1]: a = np.linspace(0, 1, 1E+06)

In [2]: %timeit  b = 3*a -1
100 loops, best of 3: 10.7 ms per loop

In [3]: b = np.zeros(1E+06)

In [4]: %timeit for i in xrange(a.size): b[i] = 3*a[i] - 1
10 loops, best of 3: 1.31 s per loop
```

On va 122 fois plus vite en vectorisant ! !

Calculs sur les tableaux

```
In [1]: def f1(x):
   ...:     return np.exp(-x*x)*np.log(1+x*np.sin(x))
   ...:

In [2]: x = np.linspace(0, 1, 1e6)

In [3]: a = f1(x)

In [4]: %timeit a = f1(x)
10 loops, best of 3: 135 ms per loop
```

flatten

```
>>> a = np.array([[1, 2, 4], [5, 6, 9]])
>>> a
array([[1, 2, 4],
       [5, 6, 9]])
>>> a.flatten()
array([1, 2, 4, 5, 6, 9])
```

Attention, c'est une copie ! !

dot et tensordot

```
>>> a, b = np.array([[1, 2], [3, 4]]), array([2, 2])
>>> a*b
array([[2, 4],
       [6, 8]])
>>> np.dot(a, b)
array([ 6, 14])
>>> np.tensordot(a, b, axes=0)
array([[[2, 2],
        [4, 4]],

       [[6, 6],
        [8, 8]]])
>>> np.tensordot(a, b, axes=1)
array([ 6, 14])
```

mgrid et ogrid

```
>>> np.mgrid[0:2, 0:2]
array([[[0, 0],
        [1, 1]],

       [[0, 1],
        [0, 1]]])
>>> np.ogrid[0:2, 0:2]
[array([[0],
       [1]]), array([[0, 1]])]
```

meshgrid

```
>>> x, y = np.meshgrid(np.linspace(0, 2, 3),
...                    np.linspace(0, 2, 3))
>>> x
array([[ 0.,  1.,  2.],
       [ 0.,  1.,  2.],
       [ 0.,  1.,  2.]])
>>> y
array([[ 0.,  0.,  0.],
       [ 1.,  1.,  1.],
       [ 2.,  2.,  2.]])
```

all, any et where

```
>>> a = np.array([[1, 2, 4], [5, 6, 9]])
>>> np.any(a > 2)
True
>>> np.all(a > 2)
False
>>> np.where(a <= 1)
(array([0]), array([0]))
>>> np.where(a > 1)
(array([0, 0, 1, 1, 1]), array([1, 2, 0, 1, 2]))
```

Autres opérations sur un tableau a

- **a.argmax** : renvoie un tableau d'indices des valeurs maximales selon un axe
- **a.max** : renvoie le maximum
- **a.astype** : renvoie un tableau de a convertit sous un certain type
- **a.conj** : renvoie le conjugué de a
- **a.sum** : renvoie la somme des éléments de a
- **a.prod** : renvoie le produit des éléments de a
- **a.transpose** : renvoie la transposé de a
- ...

matrix

Caractéristiques

- `matrix` est une sorte de `ndarray`.
- Sa dimension est toujours 2.
- L'extraction d'une partie d'une `matrix` est encore une `matrix`.

```
>>> b = np.matrix(np.arange(9).reshape(3, 3))
>>> b
matrix([[0, 1, 2],
        [3, 4, 5],
        [6, 7, 8]])
>>> u = np.ones((3, 1))
>>> b*u
matrix([[  3.],
        [ 12.],
        [ 21.]])
```

Le module *linalg*

- calcul de norme,
- calcul de l'inverse d'une matrice creuse,
- résolution d'un système linéaire,
- calcul du déterminant,
- calcul des valeurs propres,
- ...

Généralités

- format numpy
  1. *.npy* : un tableau est retourné.
  2. *.npz* : un dictionnaire `{filename: array}` est retourné pour chaque fichier de l'archive.
- pickle
- HDF5, NetCDF, VTK, ...

load et save

```
>>> np.save('/tmp/123', np.array([[1, 2, 3], [4, 5, 6]]))
>>> np.load('/tmp/123.npy')
array([[1, 2, 3],
       [4, 5, 6]])
```

load et savez

```
>>> x = np.linspace(0, 2*pi, 10)
>>> y = np.sin(x)
>>> np.savez('/tmp/sin', x1=x, y1=y)
>>> t = np.load('/tmp/sin.npz')
>>> t.files
['y1', 'x1']
>>> t['x1']
array([ 0.        ,  0.6981317 ,  1.3962634 ,  2.0943951 ,  2.
        3.4906585 ,  4.1887902 ,  4.88692191,  5.58505361,  6.
```

loadtxt

```
>>> from StringIO import StringIO
>>> c = StringIO("0 1\n2 3")
>>> np.loadtxt(c)
array([[ 0.,  1.],
       [ 2.,  3.]])
>>> c = StringIO("1,0,2\n3,0,4")
>>> x, y = np.loadtxt(c, delimiter=',', usecols=(0, 2),
...                   unpack=True)
>>> x
array([ 1.,  3.])
>>> y
array([ 2.,  4.])
```

savetxt

```
>>> x = y = z = np.arange(0.0, 5.0, 1.0)
>>> np.savetxt('test.out', x, delimiter=',')
>>> np.savetxt('test.out', (x, y, z))
>>> np.savetxt('test.out', x, fmt='%1.4e')
```

Remarque : tous les exemples d'entrées-sorties ont été pris dans le guide de référence de numpy (voir ici).

Options pour l'affichage des tableaux

```
>>> a  = np.sin(np.arange(3))
>>> a
array([ 0.        ,  0.84147098,  0.90929743])}
>>> np.set_printoptions(precision=2, threshold=sys.maxint)
>>> a
array([ 0.  ,  0.84,  0.91])
```

Comment optimiser numpy

- utiliser l'API C directement,
- utiliser Swig,
- utiliser f2py,
- utiliser cython,
- ...

Ressources

Ressources

Ressources générales

1. site officiel www.python.org
2. Apprendre à programmer avec Python

Ressources pour le calcul scientifique

1. site de Numpy.
2. Hans P. Langtangen, *Python Scripting for Computational Science*, Edition Springer, 2004.
3. Hans P. Langtangen, *A Primer on Scientific Programming with Python*, Edition Springer, 2009.