

# Les bases du langage Python

Loïc Gouarin

Laboratoire de mathématiques d'Orsay

6 décembre 2010

# Plan

- 1 Présentation de Python
- 2 Les types et les opérations de base
- 3 Les structures de contrôle
- 4 Les fonctions
- 5 Les fichiers
- 6 Les classes
- 7 Les exceptions
- 8 Les modules
- 9 Ressources

## Plan

- 1 Présentation de Python
- 2 Les types et les opérations de base
- 3 Les structures de contrôle
- 4 Les fonctions
- 5 Les fichiers
- 6 Les classes
- 7 Les exceptions
- 8 Les modules
- 9 Ressources

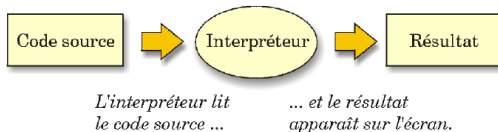
## Le langage Python

- 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

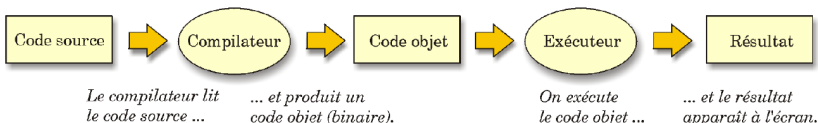
## Comment faire fonctionner mon code source ?

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

- Interprétation



- Compilation



0. figures tirées du livre "Apprendre à programmer avec Python" >

## Et Python ?



### Avantages :

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

### Inconvénients :

- peut être lent.

0. figures tirées du livre "Apprendre à programmer avec Python" >

## 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*

- **CLPython**

*Implémentation de Python en Common Lisp*

## Les différentes versions

- Il existe 2 versions de Python : 2.7 et 3.1.
- Python 3.x n'est pas une simple amélioration ou extension de Python 2.x.
- Tant que les auteurs de bibliothèques n'auront pas effectué la migration, les deux versions devront coexister.
- Nous nous intéresserons uniquement à Python 2.x.

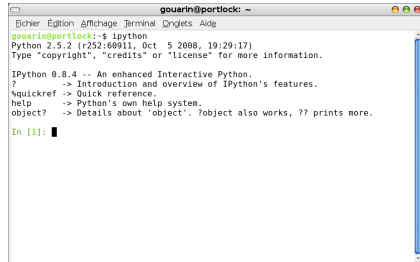


## L'interpréteur

### Sous Linux



```
gouarin@portlock: ~  
gouarin@portlock:~$ python  
Python 2.5.2 (r252:60911, Oct 5 2008, 19:29:17)  
[GCC 4.3.2] on linux2  
Type "help", "copyright", "credits" or "license" for more information.  
>>> 2 + 2  
4  
>>>
```



```
gouarin@portlock: ~  
gouarin@portlock:~$ ipython  
Python 2.5.2 (r252:60911, Oct 5 2008, 19:29:17)  
Type "copyright", "credits" or "license" for more information.  
  
IPython 0.8.4 -- An enhanced Interactive Python.  
?          -> Introduction and overview of IPython's features.  
%quickref  -> Quick reference.  
help       -> Python's own help system.  
object?    -> Details about 'object'. ?object also works, ?? prints more.  
  
In [1]: █
```

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

## Options utiles de l'interpréteur classique

- -c : exécute la commande Python entrée après,
- -i : passe en mode interactif 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
- très bon support pour le calcul scientifique

## Plan

- 1 Présentation de Python
- 2 Les types et les opérations de base**
- 3 Les structures de contrôle
- 4 Les fonctions
- 5 Les fichiers
- 6 Les classes
- 7 Les exceptions
- 8 Les modules
- 9 Ressources

# Les types et les opérations de base

- les nombres et les booléens
- les chaînes de caractères
- les listes
- les tuples
- les dictionnaires

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

### 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`



## Manipulations de chaînes de caractères

### Définir une chaîne

```
>>> "je suis une chaine"  
'je suis une chaine'  
>>> 'je suis une chaine'  
'je suis une chaine'  
>>> "j'ai bien compris"  
"j'ai bien compris"  
>>> 'J\'ai toujours la meme chose'  
"J'ai toujours la meme chose"  
>>> ""  
... je suis  
... une chaine  
... sur plusieurs  
... lignes ... ""  
'je suis\nune chaine\nsur plusieurs\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)
```

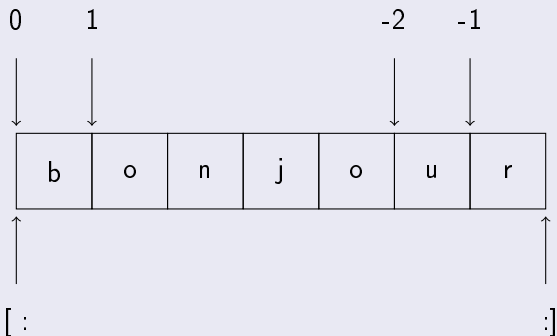
```
i vaut 1
```

```
>>> print '*-'*5
```

```
*-*-*-*--
```

## Manipulations de chaînes de caractères

### Accès au caractères [debut : fin : pas]



## Manipulations de chaînes de caractères

### Accès au 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 immuable.

## 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

### 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 élément 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
- ...

## Plan

- 1 Présentation de Python
- 2 Les types et les opérations de base
- 3 Les structures de contrôle**
- 4 Les fonctions
- 5 Les fichiers
- 6 Les classes
- 7 Les exceptions
- 8 Les modules
- 9 Ressources



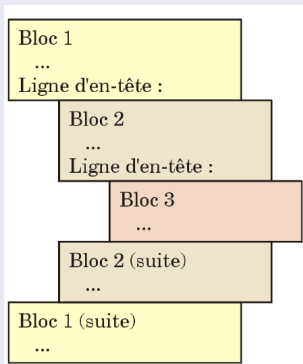
## Un petit exemple

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

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 négatif'  
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)
```

## Plan

- 1 Présentation de Python
- 2 Les types et les opérations de base
- 3 Les structures de contrôle
- 4 Les fonctions**
- 5 Les fichiers
- 6 Les classes
- 7 Les exceptions
- 8 Les modules
- 9 Ressources

## 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)
```

## Plan

- 1 Présentation de Python
- 2 Les types et les opérations de base
- 3 Les structures de contrôle
- 4 Les fonctions
- 5 Les fichiers**
- 6 Les classes
- 7 Les exceptions
- 8 Les modules
- 9 Ressources

## 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 (`\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 `\n`
- **f.write(s)** : écrit la chaîne s dans le fichier de f
- **f.writelines(lst)** : écrit la liste de chaîne lst dans le fichier de f

## Plan

- 1 Présentation de Python
- 2 Les types et les opérations de base
- 3 Les structures de contrôle
- 4 Les fonctions
- 5 Les fichiers
- 6 Les classes**
- 7 Les exceptions
- 8 Les modules
- 9 Ressources



## Définition

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

### Objet classe

admet 2 types d'opérations :

- référencement des attributs
- instantiation

## 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

## Instance

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

## Exemple

```
x = MaClasse()
```

## 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)
```

```
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])
```

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

## Plan

- 1 Présentation de Python
- 2 Les types et les opérations de base
- 3 Les structures de contrôle
- 4 Les fonctions
- 5 Les fichiers
- 6 Les classes
- 7 Les exceptions**
- 8 Les modules
- 9 Ressources



## Définition

```
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

## Format général

```
try:
    <blocs d'instructions>
except <nom1>:
    <blocs d'instructions>
except <nom2>, <donnee>:
    <blocs d'instructions>
except (nom3, nom4):
    <blocs d'instructions>
except:
    <blocs d'instructions>
else:
    <blocs d'instructions>
finally:
    <blocs d'instructions>
```

## Exemples

```
def division(x, y):  
    try:  
        resultat = x / y  
    except ZeroDivisionError:  
        print "division par zero!"  
    else:  
        print "le resultat est", resultat  
    finally:  
        print "execution de finally"
```

## Exemples

```
>>> division(2, 1)
le resultat est 2
execution de finally
>>> division(2, 0)
division par zero!
execution de finally
>>> division("2", "1")
execution de finally
Traceback (most recent call last):
File "<stdin>", line 1, in ?
File "<stdin>", line 3, in division
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

### Déclencher une exception : raise

```
>>> try:
...     raise ZeroDivisionError
... except ZeroDivisionError:
...     print 'division par zero !'
...
division par zero !
```

### Définir ses propres exceptions

```
>>> class MonErreur(Exception):
...     def __init__(self, valeur):
...         self.valeur = valeur
...     def __str__(self):
...         return repr(self.valeur)
...
>>> try:
...     raise MonErreur(2*2)
... except MonErreur, valeur:
...     print 'Mon exception s'est produite:', valeur
...
Mon exception s'est produite: 4
```

## Plan

- 1 Présentation de Python
- 2 Les types et les opérations de base
- 3 Les structures de contrôle
- 4 Les fonctions
- 5 Les fichiers
- 6 Les classes
- 7 Les exceptions
- 8 Les modules**
- 9 Ressources

## Exemple : fibo.py

```
# Module nombres de Fibonacci
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
    print

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
```



## 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 paquets
- 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"]
```

### Utilisation

```
>>> 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.

## 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 désigne 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, ...
- ...

## Présentation de distutils

setup.py

```
from distutils.core import setup

setup(name = 'monmodule',
      version = '1.0',
      py_modules = ['monfichier'],
      )
```

## Construction du module

```
$ python setup.py build
```

### Création du répertoire build

- contient les fichiers à installer
- *lib.platforme* : modules pure Python et extensions
- *temp.platforme* : fichiers temporaires générés lors de l'utilisation d'extension.

## Installation du module

```
$ python setup.py install
```

- copie tout ce qu'il y a dans `build/lib.plateforme` dans le répertoire d'installation
- le répertoire d'installation par défaut est
  - windows : `C:\Python`
  - Unix (pure) : `/usr/local/lib/pythonX.Y/site-packages`
  - Unix (non-pure) :  
`/usr/local/lib/pythonX.Y/site-packages`

## Installation du module

### Du coté développeur

```
$ python setup.py install --home=<dir>
```

Installation dans <dir>/lib/python

### Du coté administrateur

```
$ python setup.py install --prefix=<dir>
```

Installation dans <dir>/lib/pythonX.Y/site-packages

## Plan

- 1 Présentation de Python
- 2 Les types et les opérations de base
- 3 Les structures de contrôle
- 4 Les fonctions
- 5 Les fichiers
- 6 Les classes
- 7 Les exceptions
- 8 Les modules
- 9 Ressources**

## Ressources générales

- 1 site officiel [www.python.org](http://www.python.org)
- 2 Apprendre à programmer avec Python
- 3 Plongez au coeur de Python
- 4 ...

## Ressources pour le calcul scientifique

- 1 liste de diffusion de [Numpy et Scipy](#).
- 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.
- 4 ...