

Introduction à Python

Alexandre Gramfort : alexandre.gramfort@telecom-paristech.fr

Slim Essid : slim.essid@telecom-paristech.fr

adapté du travail de J.R. Johansson (robert@riken.jp) <http://dml.riken.jp/~rob/> (<http://dml.riken.jp/~rob/>)

Installation

Linux

Sous Ubuntu Linux:

```
$ sudo apt-get install python ipython ipython-notebook
```

```
$ sudo apt-get install python-numpy python-scipy python-matplotlib python-sympy
```

```
$ sudo apt-get install spyder
```

MacOS X

- [Anaconda CE \(http://continuum.io/downloads.html\)](http://continuum.io/downloads.html)

Windows

- [Python\(x,y\) \(http://code.google.com/p/pythonxy/\)](http://code.google.com/p/pythonxy/)
- [Anaconda CE \(http://continuum.io/downloads.html\)](http://continuum.io/downloads.html) (recommandé)

Remarque

Anaconda CE est aussi disponible sous Linux

En salle de TP à Télécom ParisTech

Tapez dans un terminal:

```
$ export PATH=/cal/softs/anaconda/anaconda-latest/bin/:$PATH
```

Lancer un programme Python

- Un fichier python termine par ".py":

```
mon_programme.py
```

- Toutes les lignes d'un fichier Python sont exécutées sauf les lignes qui commencent par # **qui sont des commentaires**.
- Pour lancer le programme depuis une ligne de commande ou un terminal:

```
$ python mon_programme.py
```

- Sous UNIX (Linux / Mac OS) il est courant d'ajouter le chemin vers l'interpréteur python sur la première ligne du fichier:

```
#!/usr/bin/env python
```

Cela permet de lancer un programme directement:

```
$ mon_programme.py
```

Exemple:

```
In [1]: ls scripts/hello-world.py
scripts/hello-world.py*
```

```
In [2]: cat scripts/hello-world.py
#!/usr/bin/env python

print("Hello world!")
```

```
In [3]: !./scripts/hello-world.py
Hello world!
```

Commencer une ligne par ! dans ipython permet de lancer une commande UNIX.

L'interpréteur Python (mode interactif)

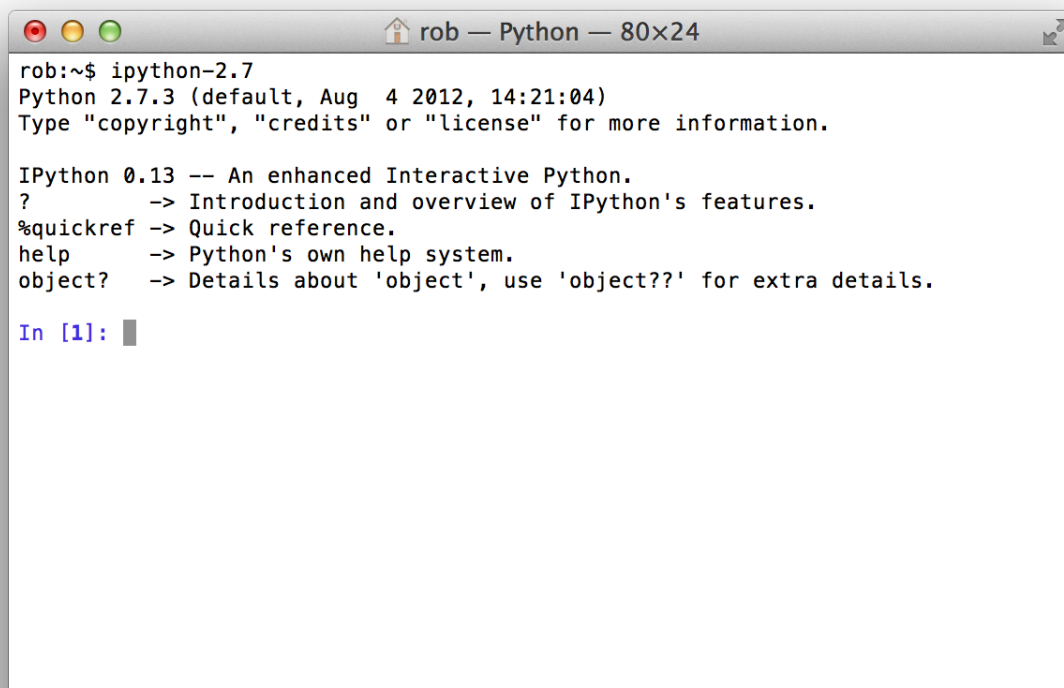
L'interpréteur Python se lance avec la commande `python`. Pour sortir taper `exit()` ou `Ctrl+D`



```
rob:~$ python
Python 2.7.2 (default, Jun 20 2012, 16:23:33)
[GCC 4.2.1 Compatible Apple Clang 4.0 (tags/Apple/clang-418.0.60)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("hello world")
hello world
>>> █
```

IPython

IPython est un shell interactif beaucoup plus avancé.

A terminal window titled "rob — Python — 80x24" showing the execution of the command "ipython-2.7". The output displays the Python version (2.7.3) and the IPython version (0.13), along with a list of help commands: "?", "%quickref", "help", and "object?". The prompt "In [1]:" is visible at the bottom.

```
rob:~$ ipython-2.7
Python 2.7.3 (default, Aug 4 2012, 14:21:04)
Type "copyright", "credits" or "license" for more information.

IPython 0.13 -- An enhanced Interactive Python.
?      -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help    -> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.

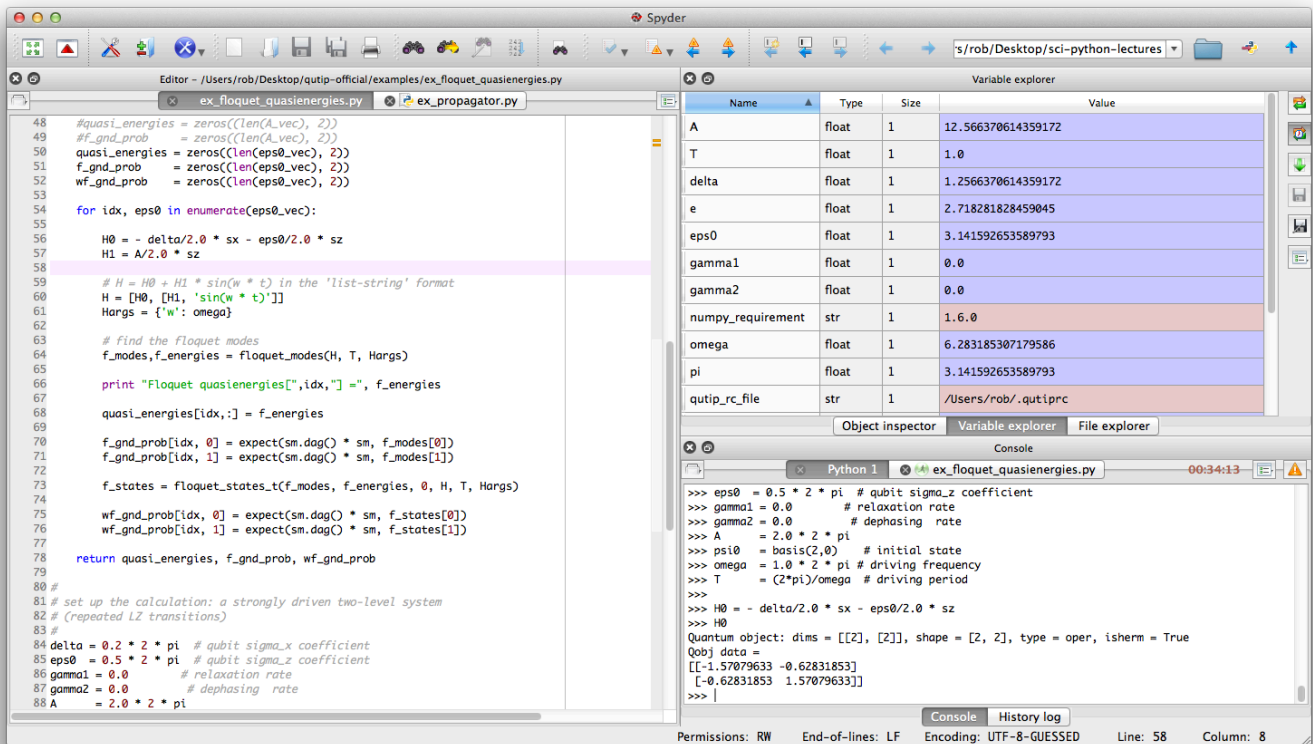
In [1]: █
```

Il permet notamment de:

- mémoriser les commandes lancées précédemment avec les flèches (haut et bas).
- auto-complétion avec Tab.
- édition de code inline
- accès simple à la doc
- debug

Spyder

Spyder (<http://code.google.com/p/spyderlib/>) est un IDE similaire à MATLAB.

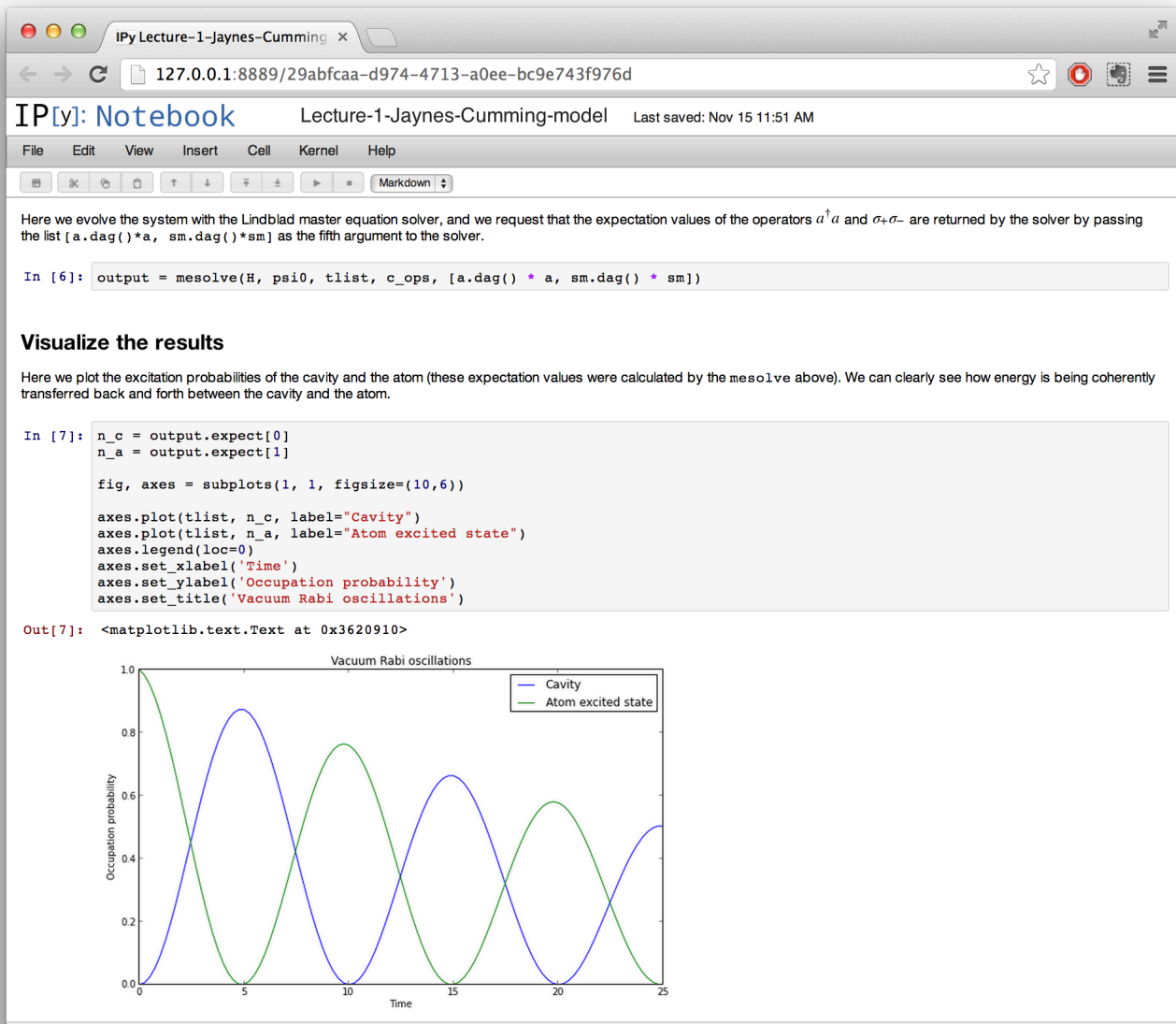


Les avantages de Spyder:

- Bon éditeur (couleurs, intégré avec le debugger).
- Explorateur de variables, intégration de IPython
- Documentation intégrée.

IPython notebook

IPython notebook (<http://ipython.org/ipython-doc/dev/interactive/htmlnotebook.html/>) comme Mathematica ou Maple dans une page web.



Pour lancer ipython-notebook:

```
$ ipython notebook
```

depuis un dossier où seront stockés les notebooks (fichiers *.ipynb)

Les nombres

```
In [4]: 2 + 2 + 1 # commentaire
```

```
Out[4]: 5
```

```
In [5]: a = 4
        print a
        print type(a)

4
<type 'int'>
```

Les noms de variable peuvent contenir a-z, A-Z, 0-9 et quelques caractères spéciaux tels que `_` mais commencent toujours par une lettre.

Par convention les noms de variables sont en minuscule.

Quelques noms de variable ne sont pas autorisés car déjà utilisés par le langage:

```
and, as, assert, break, class, continue, def, del, elif, else, except,
exec, finally, for, from, global, if, import, in, is, lambda, not, or,
pass, print, raise, return, try, while, with, yield
```

```
In [6]: int a = 1; # in C

File "<ipython-input-6-232418b2343a>", line 1
    int a = 1; # in C
          ^
SyntaxError: invalid syntax
```

```
In [7]: c = 2.1 # nombre flottant
        print type(c)

<type 'float'>
```

```
In [8]: a = 1.5 + 1j # nombre complexe
        print a.real
        print a.imag
        print 1j
        print a
        print a + 1j
        print 1j * 1j
        print type(a)

1.5
1.0
1j
(1.5+1j)
(1.5+2j)
(-1+0j)
<type 'complex'>
```

```
In [9]: type(1j * 1j)
```

```
Out[9]: complex
```

```
In [10]: 3 < 4 # bool
```

```
Out[10]: True
```

```
In [11]: 1 < 3 < 5
```

```
Out[11]: True
```

```
In [12]: 3 < 2
```

```
Out[12]: False
```

```
In [13]: test = (3 > 4)
         print test
```

```
False
```

```
In [14]: type(test)
```

```
Out[14]: bool
```

```
In [15]: print 7 * 3. # int x float -> float
         print type(7 * 3.)
```

```
21.0
```

```
<type 'float'>
```

```
In [16]: 2 ** 10 # exposant. attention pas ^
```

```
Out[16]: 1024
```

```
In [17]: 8 % 3 # reste de la division (modulo)
```

```
Out[17]: 2
```

Attention !

```
In [18]: 3 / 2 # int x int -> int
```

```
Out[18]: 1
```

```
In [19]: 3 / 2. # OK
```

```
Out[19]: 1.5
```

```
In [20]: 3 // 2
```

```
Out[20]: 1
```



```
In [21]: a = 2  
3 / float(a) # OK
```

```
Out[21]: 1.5
```

MAIS:

```
In [22]: cos(2)
```

```
-----  
-----  
NameError                                Traceback (most recent call  
last)  
<ipython-input-22-43abd96808db> in <module>()  
----> 1 cos(2)  
  
NameError: name 'cos' is not defined
```

La bibliothèque standard et ses modules

- Les fonctions Python sont organisées par *modules*
- Bibliothèque standard Python (*Python Standard Library*) : collection de modules donnant accès à des fonctionnalités de bases : appels au système d'exploitation, gestion des fichiers, gestion des chaînes de caractères, interface réseau, etc.

Références

- The Python Language Reference: <http://docs.python.org/2/reference/index.html> (<http://docs.python.org/2/reference/index.html>)
- The Python Standard Library: <http://docs.python.org/2/library/> (<http://docs.python.org/2/library/>)

Utilisation des modules

- Un module doit être *importé* avant de pouvoir être utilisé, exemple :

```
In [23]: import math
```

- Le module `math` peut maintenant être utilisé dans la suite du programme :

```
In [24]: import math

x = math.cos(2 * math.pi)

print x

1.0
```

Ou bien en important que les fonctions dont on a besoin:

```
In [25]: from math import cos, pi

x = cos(2 * pi)

print x

1.0
```

```
In [26]: from math import *

tanh(1)
```

```
Out[26]: 0.7615941559557649
```

```
In [27]: import math as m

print m.cos(1.)

0.540302305868
```

Connaitre le contenu d'un module

- Une fois un module importé on peut lister les symboles disponibles avec la fonction `dir`:

```
In [28]: import math

print dir(math)

['__doc__', '__file__', '__name__', '__package__', 'acos', 'acosh',
'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos',
'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'fa
ctorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isinf',
'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'modf', 'pi',
'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

- Pour accéder à l'aide : `help`

```
In [29]: help(math.log)
```

```
Help on built-in function log in module math:
```

```
log(...)  
    log(x[, base])
```

```
    Return the logarithm of x to the given base.  
    If the base not specified, returns the natural logarithm (base e  
    ) of x.
```

- Dans Spyder ou IPython, on peut faire:

```
In [30]: math.log?
```

```
In [31]: math.log(10)
```

```
Out[31]: 2.302585092994046
```

```
In [32]: math.log(10, 2)
```

```
Out[32]: 3.3219280948873626
```

```
In [33]: math.ceil(2.5)
```

```
Out[33]: 3.0
```

- help peut être aussi utilisée sur des modules :

```
In [34]: help(math)
```

```
Help on module math:
```

```
NAME
```

```
    math
```

```
FILE
```

```
    /Users/alex/anaconda/lib/python2.7/lib-dynload/math.so
```

```
MODULE DOCS
```

```
    http://docs.python.org/library/math
```

```
DESCRIPTION
```

```
    This module is always available. It provides access to the  
    mathematical functions defined by the C standard.
```

```
FUNCTIONS
```

```
    acos(...)  
    acos(x)
```

Return the arc cosine (measured in radians) of x .

```
acosh(...)  
acosh(x)
```

Return the inverse hyperbolic cosine of x .

```
asin(...)  
asin(x)
```

Return the arc sine (measured in radians) of x .

```
asinh(...)  
asinh(x)
```

Return the inverse hyperbolic sine of x .

```
atan(...)  
atan(x)
```

Return the arc tangent (measured in radians) of x .

```
atan2(...)  
atan2(y, x)
```

Return the arc tangent (measured in radians) of y/x .
Unlike `atan(y/x)`, the signs of both x and y are considered.

```
atanh(...)  
atanh(x)
```

Return the inverse hyperbolic tangent of x .

```
ceil(...)  
ceil(x)
```

Return the ceiling of x as a float.
This is the smallest integral value $\geq x$.

```
copysign(...)  
copysign(x, y)
```

Return x with the sign of y .

```
cos(...)  
cos(x)
```

Return the cosine of x (measured in radians).

```
cosh(...)  
cosh(x)
```

Return the hyperbolic cosine of x .

```
degrees(...)
degrees(x)
```

Convert angle x from radians to degrees.

```
erf(...)
erf(x)
```

Error function at x .

```
erfc(...)
erfc(x)
```

Complementary error function at x .

```
exp(...)
exp(x)
```

Return e raised to the power of x .

```
expm1(...)
expm1(x)
```

Return $\exp(x)-1$.

This function avoids the loss of precision involved in the direct evaluation of $\exp(x)-1$ for small x .

```
fabs(...)
fabs(x)
```

Return the absolute value of the float x .

```
factorial(...)
factorial(x) -> Integral
```

Find $x!$. Raise a `ValueError` if x is negative or non-integral

.

```
floor(...)
floor(x)
```

Return the floor of x as a float.

This is the largest integral value $\leq x$.

```
fmod(...)
fmod(x, y)
```

Return $\text{fmod}(x, y)$, according to platform C. $x \% y$ may differ.

```
frexp(...)
frexp(x)
```

Return the mantissa and exponent of x , as pair (m, e) .
 m is a float and e is an int, such that $x = m * 2.**e$.
If x is 0, m and e are both 0. Else $0.5 \leq \text{abs}(m) < 1.0$.

```
fsum(...)  
fsum(iterable)
```

Return an accurate floating point sum of values in the iterable.

Assumes IEEE-754 floating point arithmetic.

```
gamma(...)  
gamma(x)
```

Gamma function at x .

```
hypot(...)  
hypot(x, y)
```

Return the Euclidean distance, $\text{sqrt}(x*x + y*y)$.

```
isinf(...)  
isinf(x) -> bool
```

Check if float x is infinite (positive or negative).

```
isnan(...)  
isnan(x) -> bool
```

Check if float x is not a number (NaN).

```
ldexp(...)  
ldexp(x, i)
```

Return $x * (2**i)$.

```
lgamma(...)  
lgamma(x)
```

Natural logarithm of absolute value of Gamma function at x .

```
log(...)  
log(x[, base])
```

Return the logarithm of x to the given base.
If the base not specified, returns the natural logarithm (base e) of x .

```
log10(...)  
log10(x)
```

Return the base 10 logarithm of x .

```
loglp(...)
```

```
log1p(x)
```

Return the natural logarithm of 1+x (base e).
The result is computed in a way which is accurate for x near zero.

```
modf(...)  
modf(x)
```

Return the fractional and integer parts of x. Both results carry the sign of x and are floats.

```
pow(...)  
pow(x, y)
```

Return x^y (x to the power of y).

```
radians(...)  
radians(x)
```

Convert angle x from degrees to radians.

```
sin(...)  
sin(x)
```

Return the sine of x (measured in radians).

```
sinh(...)  
sinh(x)
```

Return the hyperbolic sine of x.

```
sqrt(...)  
sqrt(x)
```

Return the square root of x.

```
tan(...)  
tan(x)
```

Return the tangent of x (measured in radians).

```
tanh(...)  
tanh(x)
```

Return the hyperbolic tangent of x.

```
trunc(...)  
trunc(x:Real) -> Integral
```

Truncates x to the nearest Integral toward 0. Uses the `__trunc__` magic method.

```
DATA
e = 2.718281828459045
pi = 3.141592653589793
```

- Modules utiles de bibliothèque standard: `os`, `sys`, `math`, `shutil`, `re`, etc.
- Pour une liste complète, voir: <http://docs.python.org/2/library/> (<http://docs.python.org/2/library/>)

EXERCICE : Ecrire un code qui calcule la première puissance de 2 supérieure à un nombre n

```
In [35]: import math as m
# à compléter
```

Fractions

```
In [36]: import fractions
a = fractions.Fraction(2, 3)
b = fractions.Fraction(1, 2)
print(a + b)
```

```
7/6
```

- On peut utiliser `isinstance` pour tester les types des variables :

```
In [37]: print type(a)
print isinstance(a, fractions.Fraction)
```

```
<class 'fractions.Fraction'>
True
```

```
In [38]: a = fractions.Fraction(1, 1)
print(a)
print(type(a))
```

```
1
<class 'fractions.Fraction'>
```

Type casting (conversion de type)


```
In [39]: x = 1.5
         print x, type(x)
1.5 <type 'float'>
```

```
In [40]: x = int(x)
         print x, type(x)
1 <type 'int'>
```

```
In [41]: z = complex(x)
         print z, type(z)
(1+0j) <type 'complex'>
```

```
In [42]: x = float(z)
         print x, type(x)

-----
-----
TypeError                                 Traceback (most recent call
last)
<ipython-input-42-b47ba47d3a3d> in <module>()
----> 1 x = float(z)
      2 print x, type(x)

TypeError: can't convert complex to float
```

Operateurs et comparaisons

```
In [43]: 1 + 2, 1 - 2, 1 * 2, 1 / 2  # + - / * sur des entiers
```

```
Out[43]: (3, -1, 2, 0)
```

```
In [44]: 1.0 + 2.0, 1.0 - 2.0, 1.0 * 2.0, 1.0 / 2.0  # + - / * sur des flott
ants
```

```
Out[44]: (3.0, -1.0, 2.0, 0.5)
```

```
In [45]: # Division entière
         3.0 // 2.0
```

```
Out[45]: 1.0
```

```
In [46]: # Attention ** et pas ^
         2 ** 2
```

```
Out[46]: 4
```

- Opérations booléennes en anglais `and`, `not`, `or`.

```
In [47]: True and False
```

```
Out[47]: False
```

```
In [48]: not False
```

```
Out[48]: True
```

```
In [49]: True or False
```

```
Out[49]: True
```

- Comparisons `>`, `<`, `>=` (plus grand ou égal), `<=` (inférieur ou égal), `==` égalité, `is` identique.

```
In [50]: 2 > 1, 2 < 1
```

```
Out[50]: (True, False)
```

```
In [51]: 2 > 2, 2 < 2
```

```
Out[51]: (False, False)
```

```
In [52]: 2 >= 2, 2 <= 2
```

```
Out[52]: (True, True)
```

```
In [53]: 2 != 3
```

```
Out[53]: True
```

```
In [54]: not 2 == 3
```

```
Out[54]: True
```

```
In [55]: int(True)
```

```
Out[55]: 1
```

```
In [56]: int(False)
```

```
Out[56]: 0
```

```
In [57]: 1 == True
```

```
Out[57]: True
```

Conteneurs: Chaînes de caractères, listes et dictionnaires

Chaines de caractères (Strings)

```
In [58]: s = 'Hello world!'
# ou avec " "
s = "Hello world!"
print s
print type(s)
```

```
Hello world!
<type 'str'>
```

Attention: les indices commencent à 0!

On peut extraire une sous-chaine avec la syntaxe [start:stop], qui extrait les caractères entre start et stop (**exclu**) :

```
In [59]: s[0] # premier élément
```

```
Out[59]: 'H'
```

```
In [60]: s[-1] # dernier élément
```

```
Out[60]: '!'
```

```
In [61]: s[1:5]
```

```
Out[61]: 'ello'
```

```
In [62]: start, stop = 1, 5
print s[start:stop]
print len(s[start:stop])
```

```
ello
4
```

```
In [63]: print stop - start
```

```
4
```

```
In [64]: print start
print stop
```

```
1
5
```

Attention car :

```
In [65]: len("é")
```

```
Out[65]: 2
```

Mais:

```
In [66]: len(u"é") # chaine de caractères unicode
```

```
Out[66]: 1
```

On peut omettre `start` ou `stop`. Dans ce cas les valeurs par défaut sont respectivement 0 et la fin de la chaîne.

```
In [67]: s[:5] # 5 premières valeurs
```

```
Out[67]: 'Hello'
```

```
In [68]: s[6:] # de l'entrée d'indice 6 à la fin
```

```
Out[68]: 'world!'
```

```
In [69]: print(len(s[6:]))  
print(len(s) - 6)
```

```
6  
6
```

```
In [70]: s[-6:] # les 6 derniers
```

```
Out[70]: 'world!'
```

Il est aussi possible de définir le `step` (pas d'avancement) avec la syntaxe `[start:stop:step]` (la valeur par défaut de `step` est 1):

```
In [71]: s[1::2]
```

```
Out[71]: 'el ol!'
```

```
In [72]: s[0::2]
```

```
Out[72]: 'Hlowrd'
```

Cette technique est appelée *slicing*. Pour en savoir plus:

<http://docs.python.org/release/2.7.3/library/functions.html?highlight=slice#slice>

(<http://docs.python.org/release/2.7.3/library/functions.html?highlight=slice#slice>) et

<http://docs.python.org/2/library/string.html> (<http://docs.python.org/2/library/string.html>)

EXERCICE : A partir des lettres de l'alphabet, générer par une operation de slicing la chaîne de caractère *cfilorux*

```
In [73]: import string
alphabet = string.ascii_lowercase
```

Mise en forme de chaînes de caractères

```
In [74]: print "str1", "str2", "str3" # print ajoute des espaces entre les chaînes
str1 str2 str3
```

```
In [75]: print "str1", 1.0, False, -1j # print convertit toutes les variables en chaînes
str1 1.0 False -1j
```

```
In [76]: print "str1" + "str2" + "str3" # pour concatener +
str1str2str3
```

```
In [77]: print "str1" * 3
str1str1str1
```

```
In [78]: a = 1.0000000002
print "val = %e" % a # comme en C (cf. printf)
print "val = %1.15f" % a # comme en C (cf. printf)
print "val = % 3d" % 10 # comme en C (cf. printf)
print "val = %s" % a # comme en C (cf. printf)
print str(a)
print "val = " + str(a)

val = 1.000000e+00
val = 1.000000000200000
val = 10
val = 1.0000000002
1.0000000002
val = 1.0000000002
```

```
In [79]: # Plus avancé
s = "val1 = %.2f, val2 = %d" % (3.1415, 1.5)
print s

val1 = 3.14, val2 = 1
```

```
In [81]: s = "Le nombre %s est égal à %s"
print(s % ("pi", math.pi))
print(s % ("e", math.exp(1.)))

Le nombre pi est égal à 3.14159265359
Le nombre e est égal à 2.71828182846
```

Listes

Les listes sont très similaires aux chaînes de caractères sauf que les éléments peuvent être de n'importe quel type.

La syntaxe pour créer des listes est [...]:

```
In [82]: l = [1, 2, 3, 4]
print type(l)
print l

<type 'list'>
[1, 2, 3, 4]
```

Exemples de slicing:

```
In [83]: print l[1:3]
print l[::2]

[2, 3]
[1, 3]
```

Attention: On commence à indexer à 0!

```
In [84]: l[0]
```

```
Out[84]: 1
```

```
In [85]: l[-1:0:-1]
```

```
Out[85]: [4, 3, 2]
```

On peut mélanger les types:

```
In [86]: l = [1, 'a', 1.0, 1-1j]
         print l
         [1, 'a', 1.0, (1-1j)]
```

On peut faire des listes de listes (par exemple pour décrire un arbre...)

```
In [87]: list_of_list = [1, [2, [3, [4, [5]]]]]
         list_of_list
```

```
Out[87]: [1, [2, [3, [4, [5]]]]]
```

```
In [88]: arbre = [1, [2, 3]]
         print arbre
         [1, [2, 3]]
```

La fonction range pour générer une liste d'entiers:

```
In [89]: start, stop, step = 10, 30, 2
         print(range(start, stop, step))
         print(range(10, 30, 2))
         print(list(range(10, 30, 2)))
         [10, 12, 14, 16, 18, 20, 22, 24, 26, 28]
         [10, 12, 14, 16, 18, 20, 22, 24, 26, 28]
         [10, 12, 14, 16, 18, 20, 22, 24, 26, 28]
```

Intération de n-1 à 0

```
In [90]: n = 10
         print(range(n-1, -1, -1))
         [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

```
In [91]: type(range(-10, 10))
```

```
Out[91]: list
```

```
In [92]: # convertir une chaine de caractère en liste
         s = "zabcda"
         l2 = list(s)
         print(l2)
         ['z', 'a', 'b', 'c', 'd', 'a']
```

```
In [93]: # tri
13 = list(l2)
l2.sort()
print(l2)
print(l3)
print(l2[::-1]) # avec copie

['a', 'a', 'b', 'c', 'd', 'z']
['z', 'a', 'b', 'c', 'd', 'a']
['z', 'd', 'c', 'b', 'a', 'a']
```

Attention l2.sort() ne renvoie rien c'est-à-dire None

```
In [94]: out = l2.sort()
print(out)
```

None

Pour renvoyer une nouvelle liste triée:

```
In [95]: l2 = list(s)
print(l2)
out = sorted(l2)
print(out)

['z', 'a', 'b', 'c', 'd', 'a']
['a', 'a', 'b', 'c', 'd', 'z']
```

Ajout, insertion, modifier, et enlever des éléments d'une liste:

```
In [96]: # création d'une liste vide
l = [] # ou l = list()

# ajout d'éléments avec `append`
m = l.append("A")
l.append("d")
l.append("d")

print(m)
print(l)
```

None

['A', 'd', 'd']

Concatenation de listes avec +


```
In [97]: l
```

```
Out[97]: ['A', 'd', 'd']
```

```
In [98]: l.index('A')
```

```
Out[98]: 0
```

```
In [99]: lll = [1, 2, 3]
mmm = [4, 5, 6]
print(lll + mmm) # attention différent de lll.append(mmm)

[1, 2, 3, 4, 5, 6]
```

```
In [100]: print(lll * 2)
```

```
[1, 2, 3, 1, 2, 3]
```

On peut modifier une liste par assignation:

```
In [101]: l[1] = "p"
l[2] = "p"
print(l)
```

```
['A', 'p', 'p']
```

```
In [102]: l[1:3] = ["d", "d"]
print(l)
```

```
['A', 'd', 'd']
```

Insertion à un index donné avec insert

```
In [103]: l.insert(0, "i")
l.insert(1, "n")
l.insert(2, "s")
l.insert(3, "e")
l.insert(4, "r")
l.insert(5, "t")
```

```
print(l)
```

```
['i', 'n', 's', 'e', 'r', 't', 'A', 'd', 'd']
```

Suppression d'un élément avec 'remove'

```
In [104]: l.remove("A")
          print(l)
          ['i', 'n', 's', 'e', 'r', 't', 'd', 'd']
```

```
In [105]: ll = [1, 2, 3, 2]
          print(ll)
          ll.remove(2)
          print(ll)
          [1, 2, 3, 2]
          [1, 3, 2]
```

```
In [106]: print(2 in ll)
          print(ll.index(2))
          True
          2
```

Suppression d'un élément à une position donnée avec del:

```
In [107]: del l[7]
          del l[6]
          print(l)
          ['i', 'n', 's', 'e', 'r', 't']
```

help(list) pour en savoir plus.

Tuples

- Les *tuples* (n-uplets) ressemblent aux listes mais ils sont *immuables* : ils ne peuvent pas être modifiés une fois créés.
- On les crée avec la syntaxe (... , ... , ...) ou simplement ..., ...:

```
In [108]: point = (10, 20)
          print(point, type(point))
          ((10, 20), <type 'tuple'>)
```

```
In [109]: point[0]
```

```
Out[109]: 10
```

```
In [110]: p2 = list(point)
          p2[1] = 5
          print(point, p2)

          ((10, 20), [10, 5])
```

Un *tuple* peut être dépillé par assignation à une liste de variables séparées par des virgules :

```
In [111]: x, y = point

          print("x =", x)
          print("y =", y)

          ('x =', 10)
          ('y =', 20)
```

On ne peut pas faire :

```
In [112]: point[0] = 20

-----
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-112-ac1c641a5dca> in <module>()
----> 1 point[0] = 20

TypeError: 'tuple' object does not support item assignment
```

Dictionnaires

Ils servent à stocker des données de la forme *clé-valeur*.

La syntaxe pour les dictionnaires est {key1 : valeur1, ...}:

```
In [113]: params = {"parameter1" : 1.0,
                   "parameter2" : 2.0,
                   "parameter3" : 3.0}

          # ou bien

          params = dict(parameter1=1.0, parameter2=2.0, parameter3=3.0)

          print(type(params))
          print(params)

          <type 'dict'>
          {'parameter1': 1.0, 'parameter3': 3.0, 'parameter2': 2.0}
```

```
In [114]: d = {0: 'aqdf', 1:"qsdf"}
```

```
In [115]: d[0]
```

```
Out[115]: 'aqdf'
```

```
In [116]: notes = {"dupont": 15, "durant":8}
print(notes["dupont"])
notes["dupont"] = 20
del notes["dupont"]
print(notes)
print("dupont" in notes)
```

```
15
{'durant': 8}
False
```

```
In [117]: print("parameter1 =", params["parameter1"])
print("parameter2 =", params["parameter2"])
print("parameter3 =", params["parameter3"])
```

```
('parameter1 =', 1.0)
('parameter2 =', 2.0)
('parameter3 =', 3.0)
```

```
In [118]: params["parameter1"] = "A"
params["parameter2"] = "B"
```

```
# ajout d'une entrée
params["parameter4"] = "D"
```

```
print("parameter1 =", params["parameter1"])
print("parameter2 =", params["parameter2"])
print("parameter3 =", params["parameter3"])
print("parameter4 =", params["parameter4"])
```

```
('parameter1 =', 'A')
('parameter2 =', 'B')
('parameter3 =', 3.0)
('parameter4 =', 'D')
```

Suppression d'une clé:

```
In [119]: del params["parameter4"]
print(params)
```

```
{'parameter1': 'A', 'parameter3': 3.0, 'parameter2': 'B'}
```

Test de présence d'une clé

```
In [120]: "parameter1" in params
```

```
Out[120]: True
```

```
In [121]: "parameter6" in params
```

```
Out[121]: False
```

```
In [122]: params["parameter6"]
```

```
-----  
-----  
KeyError                                Traceback (most recent call  
last)  
<ipython-input-122-9d26f4da51fe> in <module>()  
----> 1 params["parameter6"]  
  
KeyError: 'parameter6'
```

Conditions, branchements et boucles

Branchements: if, elif, else

```
In [123]: a = 3  
          b = 2  
  
          if a < b:  
              print "a est strictement inférieur à b"  
          elif a > b:  
              print "b est strictement inférieur à a"  
          else:  
              print "b est égal à a"
```

```
b est strictement inférieur à a
```

```
In [124]: statement1 = a < b  
          statement2 = a > b  
  
          if statement1 is True:  
              print "a est strictement inférieur à b"  
          elif statement2 is True:  
              print "b est strictement inférieur à a"  
          else:  
              print "b est égal à a"
```

```
b est strictement inférieur à a
```

```
In [125]: statement1 = False
statement2 = False

if statement1:
    print("statement1 is True")
elif statement2:
    print("statement2 is True")
else:
    print("statement1 and statement2 are False")

statement1 and statement2 are False
```

En Python l'**indentation est obligatoire** car elle influence l'exécution du code

Examples:

```
In [126]: statement1 = True
statement2 = True

if statement1:
    if statement2:
        print("both statement1 and statement2 are True")

both statement1 and statement2 are True
```

```
In [127]: print (2 == 2) == 1
print 2 == 2 == 1

True
False
```

```
In [128]: # Mauvaise indentation!
if statement1:
    if statement2:
        print("both statement1 and statement2 are True")

File "<ipython-input-128-0516d1e4e779>", line 4
    print "both statement1 and statement2 are True"
    ^
IndentationError: expected an indented block
```

```
In [129]: statement1 = True

if statement1:
    print("printed if statement1 is True")

    print("still inside the if block")

printed if statement1 is True
still inside the if block
```

```
In [130]: statement1 = False

         if statement1:
             print("printed if statement1 is True")

         print("still inside the if block")

still inside the if block
```

Boucles

Boucles **for**:

```
In [131]: for x in [1, 2, 3]:
         print(x)

1
2
3
```

La boucle **for** itère sur les éléments de la list fournie. Par exemple:

```
In [132]: for x in range(4): # par défaut range commence à 0
         print(x)

0
1
2
3
```

Attention `range(4)` n'inclut pas 4 !

```
In [133]: for x in range(-3,3):
         print(x)

-3
-2
-1
0
1
2
```

```
In [134]: for word in ["calcul", "scientifique", "en", "python"]:  
          print(word)
```

```
calcul  
scientifique  
en  
python
```

```
In [135]: for c in "calcul":  
          print(c)
```

```
c  
a  
l  
c  
u  
l
```

Pour itérer sur un dictionnaire::

```
In [136]: for key, value in params.items():  
          print(key, " = ", value)
```

```
('parameter1', ' = ', 'A')  
( 'parameter3', ' = ', 3.0)  
( 'parameter2', ' = ', 'B')
```

```
In [137]: for key in params:  
          print(key)
```

```
parameter1  
parameter3  
parameter2
```

Parfois il est utile d'accéder à la valeur et à l'index de l'élément. Il faut alors utiliser `enumerate`:

```
In [138]: for idx, x in enumerate(range(-3,3)):  
          print(idx, x)
```

```
(0, -3)  
(1, -2)  
(2, -1)  
(3, 0)  
(4, 1)  
(5, 2)
```


EXERCICE : Compter le nombre d'occurrences de chaque caractère dans la chaîne de caractères "HelLo WorLd!!" On renverra un dictionnaire qui à la lettre associe son nombre d'occurrences.

```
In [139]: s = "HelLo WorLd!!"
```

EXERCICE : Message codé par inversion de lettres

Ecrire un code de codage et de décodage

```
In [140]: code = {'e':'a', 'l':'m', 'o':'e'}
s = 'Hello world!'
# s_code = 'Hamme wermd!'
```

Compréhension de liste: création de liste avec for:

```
In [141]: ll = [x ** 2 for x in range(0,5)]

print(ll)

# est la version courte de :
ll = list()
for x in range(0, 5):
    ll.append(x ** 2)

print(ll)

# pour les gens qui font du caml
print(map(lambda x: x ** 2, range(5)))

[0, 1, 4, 9, 16]
[0, 1, 4, 9, 16]
[0, 1, 4, 9, 16]
```

Boucles **while**:

```
In [142]: i = 0

while i < 5:
    print(i)
    i = i + 1

print("OK")

0
1
2
3
4
OK
```

EXERCICE :

Calculer une approximation de π par la formule de Wallis

$$\pi = 2 \prod_{i=1}^{\infty} \frac{4i^2}{4i^2 - 1}$$

Fonctions

Une fonction en Python est définie avec le mot clé `def`, suivi par le nom de la fonction, la signature entre parenthèses `()`, et un `:`.

Exemples:

```
In [143]: def func0():
           print("test")
```

```
In [144]: func0()

test
```

Ajout d'une documentation (docstring):

```
In [145]: def func1(s):
           """
           Affichage d'une chaine et de sa longueur
           """
           print(s, "est de longueur", len(s))
```

```
In [146]: help(func1)
```

```
Help on function func1 in module __main__:  
  
func1(s)  
    Affichage d'une chaine et de sa longueur
```

```
In [147]: print(func1("test"))  
print(func1([1, 2, 3]))
```

```
('test', 'est de longueur', 4)  
None  
([1, 2, 3], 'est de longueur', 3)  
None
```

Retourner une valeur avec return:

```
In [148]: def square(x):  
           """  
           Retourne le carré de x.  
           """  
           return x ** 2
```

```
In [149]: print(square(4))
```

```
16
```

Retourner plusieurs valeurs:

```
In [150]: def powers(x):  
           """  
           Retourne les premières puissances de x.  
           """  
           return x ** 2, x ** 3, x ** 4
```

```
In [151]: print(powers(3))  
x2, x3, x4 = powers(3)  
print(x2, x3)  
print(type(powers(3)))  
out = powers(3)  
print(len(out))  
print(out[1])
```

```
(9, 27, 81)  
(9, 27)  
<type 'tuple'>  
3  
27
```

```
In [152]: t = (3,)
          print(t, type(t))
          ((3,), <type 'tuple'>)
```

```
In [153]: x2, x3, x4 = powers(3)
          print(x3)
          27
```

Arguments par défaut

Il est possible de fournir des valeurs par défaut aux paramètres:

```
In [154]: def myfunc(x, p=2, debug=False):
          if debug:
              print("evalue myfunc avec x =", x, "et l'exposant p =", p)
          return x**p
```

Le paramètre debug peut être omis:

```
In [155]: myfunc(5)
```

```
Out[155]: 25
```

```
In [156]: myfunc(5, 3)
```

```
Out[156]: 125
```

```
In [157]: myfunc(5, debug=True)
```

```
('evalue myfunc avec x =', 5, "et l'exposant p =", 2)
```

```
Out[157]: 25
```

On peut expliciter les noms de variables et alors l'ordre n'importe plus:

```
In [158]: myfunc(p=3, debug=True, x=7)
```

```
('evalue myfunc avec x =', 7, "et l'exposant p =", 3)
```

```
Out[158]: 343
```

Exercice: implémenter *quicksort*

La [page wikipedia \(http://en.wikipedia.org/wiki/Quicksort\)](http://en.wikipedia.org/wiki/Quicksort) décrivant l'algorithme de tri *quicksort* donne le pseudo-code suivant:

```
function quicksort('array')
  if length('array') <= 1
    return 'array'
  select and remove a pivot value 'pivot' from 'array'
  create empty lists 'less' and 'greater'
  for each 'x' in 'array'
    if 'x' <= 'pivot' then append 'x' to 'less'
    else append 'x' to 'greater'
  return concatenate(quicksort('less'), 'pivot', quicksort('greater'))
```

Transformer ce pseudo-code en code valide Python.

Des indices:

- la longueur d'une liste est donnée par `len(l)`
- deux listes peuvent être concaténées avec `l1 + l2`
- `l.pop()` retire le dernier élément d'une liste

Attention: une liste est mutable...

Il vous suffit de compléter cette ébauche:

```
In [159]: def quicksort(l1):
          # ...
          return

          quicksort([-2, 3, 5, 1, 3])
```

Classes

- Les *classes* sont les éléments centraux de la *programmation orientée objet*
- Classe: structure qui sert à représenter un objet et l'ensemble des opérations qui peuvent être effectuées sur ce dernier.

Dans Python une classe contient des *attributs* (variables) et des *méthodes* (fonctions). Elle est définie de manière analogue aux fonctions mais en utilisant le mot clé `class`. La définition d'une classe contient généralement un certain nombre de méthodes de classe (des fonctions dans la classe).

- Le premier argument d'une méthode doit être `self`: argument obligatoire. Cet objet `self` est une auto-référence.
- Certains noms de méthodes ont un sens particulier, par exemple :
 - `__init__`: nom de la méthode invoquée à la création de l'objet
 - `__str__`: méthode invoquée lorsque une représentation de la classe sous forme de chaîne de caractères est demandée, par exemple quand la classe est passée à `print`
 - voir <http://docs.python.org/2/reference/datamodel.html#special-method-names> (<http://docs.python.org/2/reference/datamodel.html#special-method-names>) pour les autres noms spéciaux

Exemple

```
In [160]: class Point(object):
           """
           Classe pour représenter un point dans le plan.
           """
           def __init__(self, x, y):
               """
               Creation d'un nouveau point en position x, y.
               """
               self.x = x
               self.y = y

           def translate(self, dx, dy):
               """
               Translate le point de dx and dy.
               """
               self.x += dx
               self.y += dy

           def __str__(self):
               return "Point: [%f, %f]" % (self.x, self.y)
```

Pour créer une nouvelle instance de la classe:

```
In [161]: p1 = Point(x=0, y=0) # appel à __init__
          print(p1.x)
          print(p1.y)
          print("%s" % p1)          # appel à la méthode __str__

0
0
Point: [0.000000, 0.000000]
```

```
In [162]: p1.translate(dx=1, dy=1)
          print(p1)
          print(type(p1))

Point: [1.000000, 1.000000]
<class '__main__.Point'>
```

Pour invoquer une méthode de la classe sur une instance p de celle-ci:

```
In [163]: p2 = Point(1, 1)

          p1.translate(0.25, 1.5)

          print(p1)
          print(p2)

Point: [1.250000, 2.500000]
Point: [1.000000, 1.000000]
```

Remarques

- L'appel d'une méthode de classe peut modifier l'état d'une instance particulière
- Cela n'affecte ni les autres instances ni les variables globales

Exceptions

- Dans Python les erreurs sont gérées à travers des "Exceptions"
- Une erreur provoque une *Exception* qui interrompt l'exécution normale du programme
- L'exécution peut éventuellement reprendre à l'intérieur d'un bloc de code `try - except`

- Une utilisation typique: arrêter l'exécution d'une fonction en cas d'erreur:

```
def my_function(arguments):

    if not verify(arguments):
        raise Exception("Invalid arguments")

    # et on continue
```

On utilise try et except pour maîtriser les erreurs:

```
try:
    # normal code goes here
except:
    # code for error handling goes here
    # this code is not executed unless the code
    # above generated an error
```

Par exemple:

```
In [164]: try:
           print("test_var")
           # genere une erreur: la variable test n'est pas définie
           print(test_var)
except:
           print("Caught an exception")

test_var
Caught an exception
```

Pour obtenir de l'information sur l'erreur: accéder à l'instance de la classe Exception concernée:

```
except Exception as e:
```

```
In [165]: try:
           print("test")
           # generate an error: the variable test is not defined
           print test
except Exception as e:
           print("Caught an exception:", e)

test
False
```


Manipuler les fichiers sur le disque

```
In [166]: import os
          print(os.path.join('~', 'work', 'src'))
          print(os.path.expanduser(os.path.join('~', 'work', 'src')))
```

~/work/src
/Users/alex/work/src

Quelques liens

- <http://www.python.org> (<http://www.python.org>) - Page Python officielle.
- <http://www.python.org/dev/peps/pep-0008> (<http://www.python.org/dev/peps/pep-0008>) - Recommandations de style d'écriture.
- <http://www.greenteapress.com/thinkpython/> (<http://www.greenteapress.com/thinkpython/>) - Un livre gratuit sur Python.
- [Python Essential Reference](http://www.amazon.com/Python-Essential-Reference-4th-Edition/dp/0672329786) (<http://www.amazon.com/Python-Essential-Reference-4th-Edition/dp/0672329786>) - Un bon livre de référence.