

Langage de programmation évolué pour BI



Patience passe
science !

Proverbe



Python

Plan

Chapitre I : Structures et instructions de base

- Introduction
- Rappel : Les types de bases
- Les opérations d'entrée/sortie
- Les structures de contrôles et itératives
- Les fonctions

Introduction

Python est un **Langage Orienté Objet**

Tout est objet : données, fonctions, modules...

Un objet **B** possède:

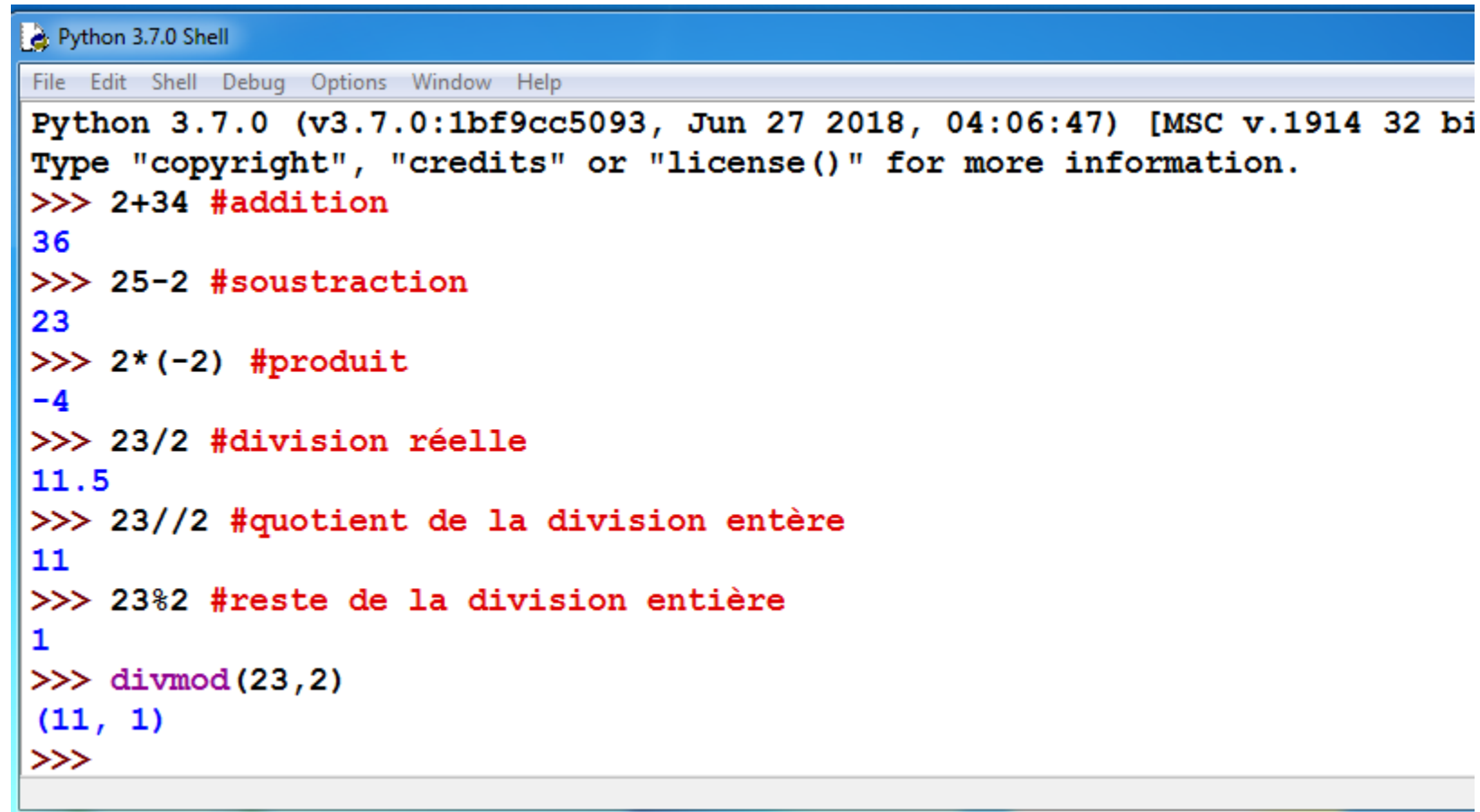
- identité id (adresse mémoire), **id(B)**
- Un type **type(B)**: (intrinsèque : int, float, str, bool ... Ou définit par l'utilisateur à l'aide d'une classe)
- contient des données (valeur).
- L'opérateur **is** compare l'identité de 2 objets (adresses)
- L'opérateur **==** compare le contenu de deux objets

Lancement de python

- Sous Windows: pour installer Python avec l'environnement de développement IDLE, il suffit de télécharger puis d'exécuter le fichier d'installation qui se trouve sur le site officiel :
<https://www.python.org/downloads/windows>
- Une fois installé, vous pouvez lancer IDLE en allant dans :
Démarrer → Programmes → Python → IDLE (Python ...)
- Une fois lancé, Python peut être utilisé en deux modes : mode interactif ou mode script

Mode interactif

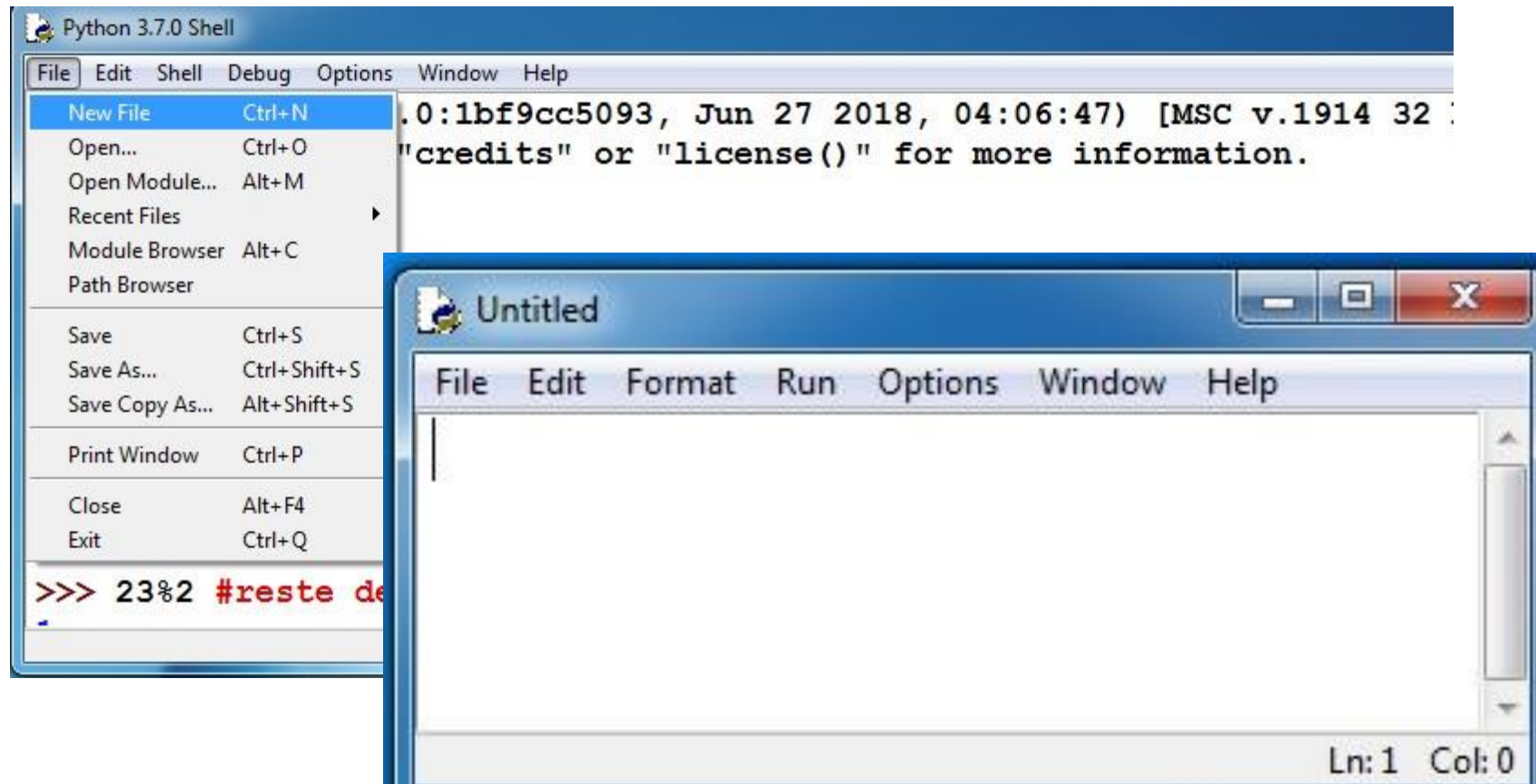
- Les instructions tapées sont exécutées directement par l'interpréteur python, c'est aussi le mode calculatrice.



```
Python 3.7.0 Shell
File Edit Shell Debug Options Window Help
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bi
Type "copyright", "credits" or "license()" for more information.
>>> 2+34 #addition
36
>>> 25-2 #soustraction
23
>>> 2*(-2) #produit
-4
>>> 23/2 #division réelle
11.5
>>> 23//2 #quotient de la division entière
11
>>> 23%2 #reste de la division entière
1
>>> divmod(23,2)
(11, 1)
>>>
```

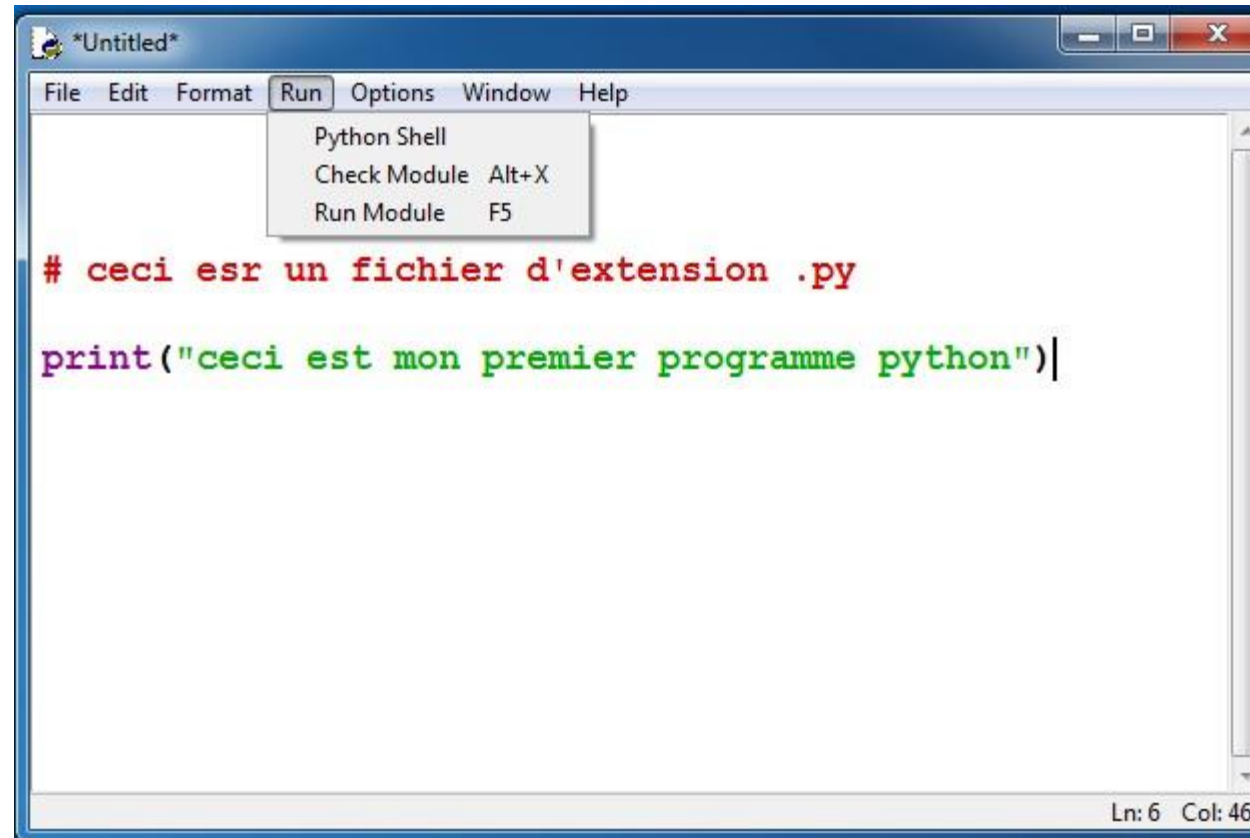
Mode script

- Les instructions tapées sont exécutées directement par l'interpréteur python, c'est aussi le mode calculatrice.



Mode script (suite)

- Ecrire le code Python (votre programme) puis lancer l'exécution (Run → Run Module (ou F5)
- Enregistrer avec un nom d'extension .py (ex : essai.py)



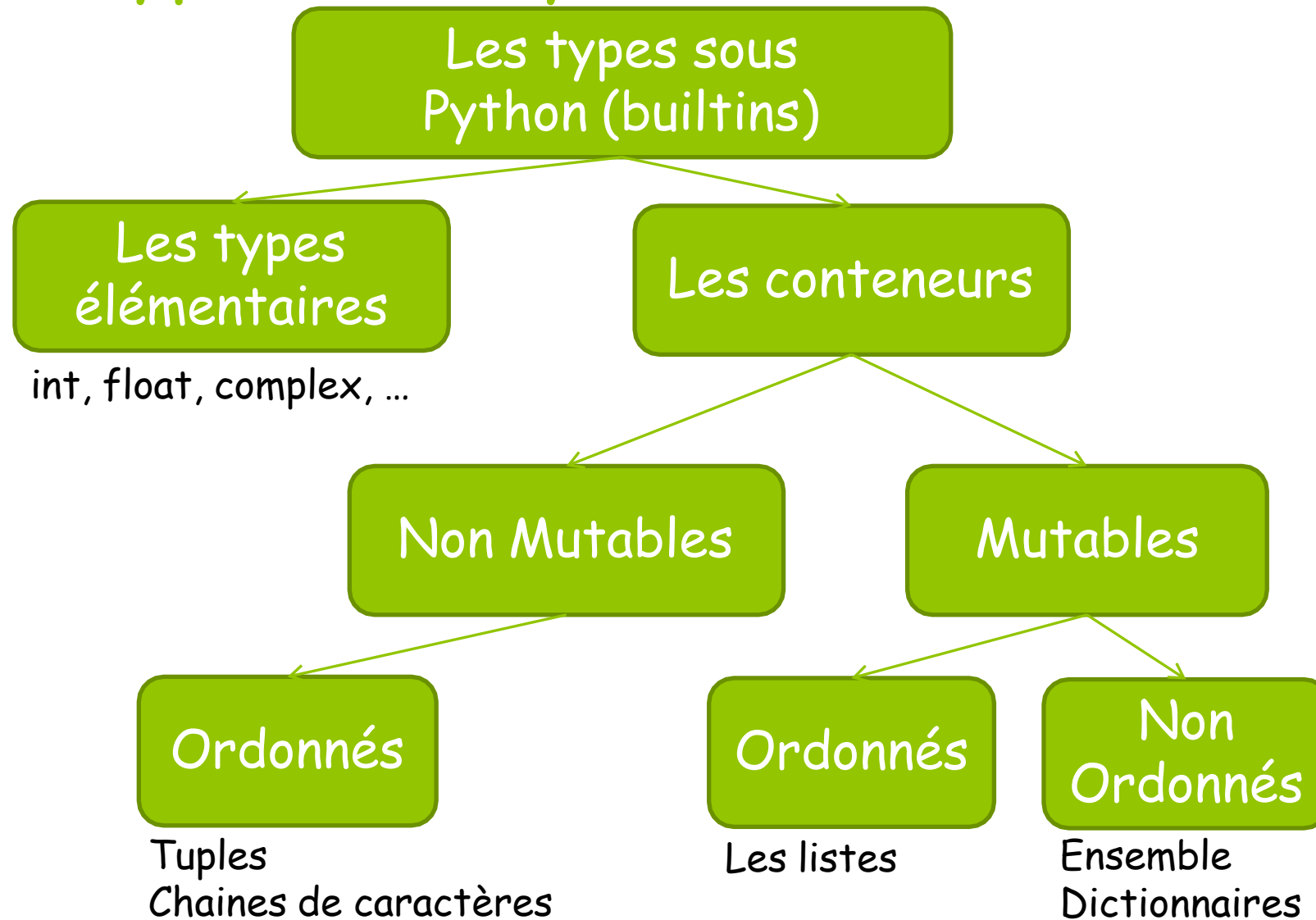
Utilisation de l'aide

- L'utilisation de l'aide en ligne se fait par la commande `help(identificateur)`
- Exemple :

```
>>> help(int)
Help on class int in module builtins:

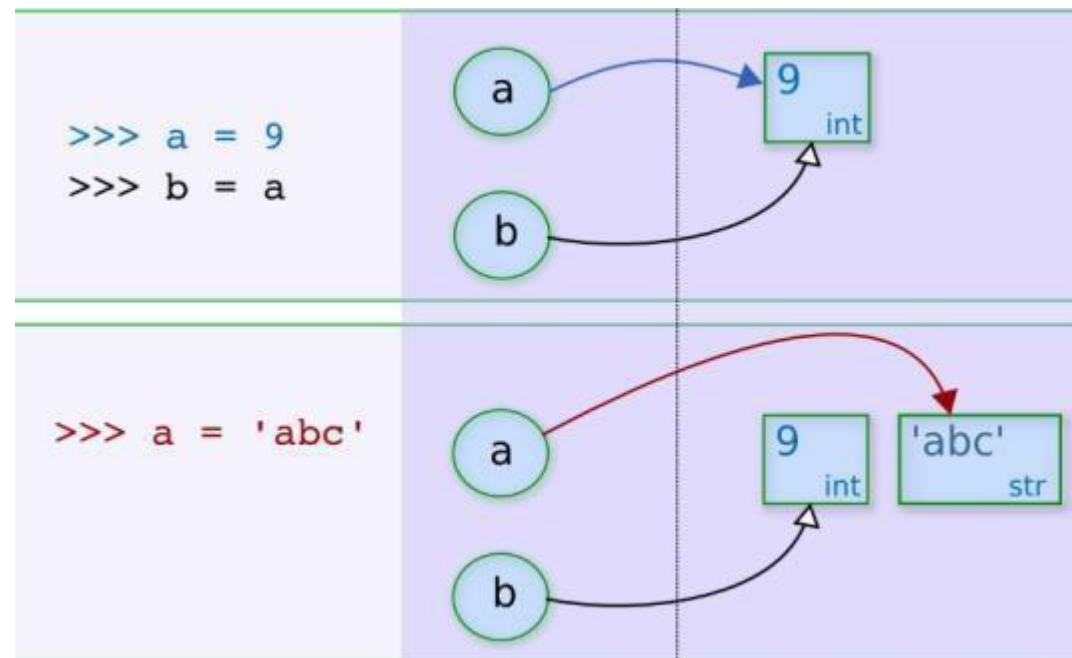
class int(object)
|   int([x]) -> integer
|   int(x, base=10) -> integer
|
|   Convert a number or string to an integer, or return 0 if no arguments
|   are given.  If x is a number, return x.__int__().  For floating point
|   numbers, this truncates towards zero.
|
|   If x is not a number or if base is given, then x must be a string,
|   bytes, or bytearray instance representing an integer literal in the
|   given base.  The literal can be preceded by '+' or '-' and be surrounded
|   by whitespace.  The base defaults to 10.  Valid bases are 0 and 2-36.
|   Base 0 means to interpret the base from the string as an integer literal.
|   >>> int('0b100', base=0)
|   4
|
|   Methods defined here:
|
|   __abs__(self, /)
|       abs(self)
|
|   __add__(self, value, /)
|       Return self+value.
|
|   __and__(self, value, /)
|       Return self&value.
```


Les types sous Python



Les types élémentaires

- Un objet ne peut changer ni d'identité ni de type !
- Quand un objet n'a plus de nom (nombre de référence nul), il est détruit automatiquement (mécanisme automatique de "ramasse miettes", ou garbage collector).



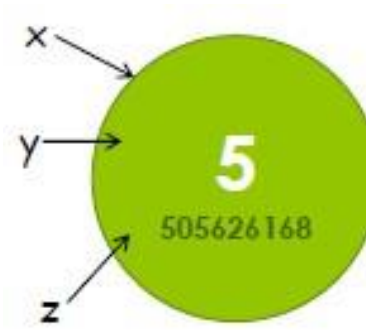
Le langage Python: L'affectation

Une affectation crée un nom (identificateur, variable) qui référence un objet (les identifiants au format `__nom__` sont réservés à l'interpréteur Python)

C'est l'objet qui porte le **type** et les données (**valeur**, pour un objet numérique).

Un même objet peut être référencé sous plusieurs noms (alias).

```
>>>x=5
>>>y=5
>>> x,id(x),type(x),
(5, 505626168, <class 'int'>)
>>>y,id(y),type(y)
(5, 505626168, <class 'int'>)
```



L'affectation (suite)

```
>>> x,y=45,23
```

```
>>> id(x),id(y)
```

```
(505626808, 505626456)
```

```
>>> x,y=y,x
```

```
>>> x,y
```

```
(23, 45)
```

```
>>> id(x),id(y)
```

```
(505626456, 505626808)
```

Les types élémentaires

- ▮ le `NoneType` : seule valeur possible `None`
c'est la valeur retournée par une fonction qui ne retourne rien
- ▮ Le type `bool` : deux valeurs possible `True` et `False` (1/0)
- ▮ Les types numériques
 - ✓ Le type `int` `x=898`
 - ✓ Le type `float` `x=8.98`
 - ✓ Le type `complex` `z=8+1j*8` `z.real`; `z.imag`; `z.conjugate()`

Les types élémentaires (suite)

Le type entier : <classe int>

```
>>> x=3
```

```
>>> y=6
```

```
>>> type(x),type(y)
```

```
(<class 'int'>, <class 'int'>)
```

Les opérations arithmétiques , + , * , ** , / , // , %

```
>>> z=x+y      #      z= x.__add__(y)
```

```
>>> z
```

```
9
```

```
>>> x-y      #      z= x.__sub__(y)
```

```
-3
```

```
>>> x*y
```

```
18
```

Les types élémentaires

```
>>> x**y # puissance
```

```
729
```

```
>>> pow(x,y)
```

```
729
```

```
>>> x/y # division réelle
```

```
0.5
```

```
>>> x//y #quotient de la division entière
```

```
0
```

```
>>> x%y #reste de la division entière
```

```
3
```

Les opérateurs
de comparaison :

<, <=, !=, ==, >, >=

```
>>> x==y
```

```
False
```

```
>>> x>y
```

```
False
```

```
>>> x<=y
```

```
True
```

```
>>> x!=y
```

```
True
```

Les types élémentaires (suite)

Le type réel: class float

```
>>> x=12/7; y=4.
```

```
>>> x;y
```

```
1.7142857142857142
```

```
4.0
```

```
>>> type(x); type(y)
```

```
<class 'float'> <class 'float'>
```

Les opérations arithmétiques

```
>>> x+y
```

```
5.714285714285714
```

```
>>> x-y
```

```
-2.2857142857142856
```

```
>>> x*y
```

```
6.857142857142857
```


Les types élémentaires (suite)

```
>>> x/y
```

```
0.42857142857142855
```

```
>>> x**y
```

```
8.636401499375259
```

```
>>> x/y
```

```
0.42857142857142855
```

```
>>> x=12/5
```

```
>>> x
```

```
2.4
```

```
>>> int(x) # Passage de réel en entier l'objet retourné est un nouvel objet
```

```
2
```

Les opérateurs de comparaison

<, <=, >, >=, ==, != de même...

Les types élémentaires (suite)

Le type booléen : class Bool

La classe Bool hérite de la classe int

```
>>> x=3 ; y=4 ; z=3
```

```
>>> B=x==y
```

```
>>> B
```

```
False
```

```
>>> E=x<y
```

```
>>> E
```

```
True
```

```
>>> B and E
```

```
False
```

```
>>> B or E
```

```
True
```

```
>>> int(True)
```

```
1
```

```
>>> int(False)
```

```
0
```

Les opérations logiques and, or, not...

Autre utilisation des opérateurs !

```
>>> x=8
```

```
>>> x+=5
```

```
>>> x
```

```
13
```

```
>>> x-=2
```

```
>>> x
```

```
11
```

```
>>> x*=3
```

```
>>> x
```

```
33
```

```
>>> x**=2
```

```
>>> x
```

```
1089
```

```
>>> x%=2
```

```
>>> x
```

```
1
```

```
>>> x/=3
```

```
>>> x
```

```
0.3333333333333333
```

```
>>> print("%5.3f"%x)
```

```
0.333
```

Trois manières d'import de modules (avantages/inconvénients)

. 1ère manière

```
>>>import math
```

```
>>>dir(math)
```

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh',  
'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1',  
'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite',  
'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians',  
'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

```
>>> math.ceil(7.8989) #partie entière supérieur
```

```
8
```

```
>>> math.floor(7.8989) #partie entière inférieur
```

```
7
```

```
>>>help(math.ceil)
```

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

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

```
    ceil(x)
```

```
        Return the ceiling of x as an int.
```

```
        This is the smallest integral value >= x.
```

. 2ème manière : Utilisation d'un alias

```
>>>import math as m
```

```
>>>dir(m)
```

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh',  
'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e',  
'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma',  
'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10',  
'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'remainder', 'sin', 'sinh', 'sqrt',  
'tan', 'tanh', 'tau', 'trunc']
```

```
>>> m.sqrt(2)
```

```
1.4142135623730951
```

```
>>> m.tan(m.pi)
```

```
-1.2246467991473532e-16
```

. 3ème manière : Importation de toutes les fonctions d'un module

```
>>>from math import *
```

```
>>>dir(math)
```

• Erreur

Traceback (most recent call last):

File "<pyshell#1>", line 1, in <module>

dir(math)

NameError: name 'math' is not defined

Mais les fonctions du module sont directement accessibles

```
>>>sqrt(2)
```

```
1.4142135623730951
```

```
>>> help(abs)
```

Help on built-in function abs in module builtins:

```
abs(...)
```

```
abs(number) -> number
```

Return the absolute value of the argument.

La troisième manière représente l'avantage d'accéder directement aux fonctions mais représente également l'inconvénient d'encombrement de l'espace de noms réservé et la possibilité de conflit entre deux fonctions ayant le même identificateur provenant de deux modules différents !!!

Exemple

```
>>>from numpy import *
```

```
>>> from math import *
```

Les deux contiennent la fonction sqrt , l'une définie pour les réels l'autre sur les tableaux! Laquelle sera utilisée ????

Les opérations d'entrée/sortie

▮ Opération d'affichage : print()

```
>>> print('ceci est un message')
```

ceci est un message

```
>>> print("ceci est un message")
```

ceci est un message

```
>>> print("ceci un message \n avec retour à la ligne")
```

ceci un message

avec retour à la ligne

```
>>> print(""" Ceci est un message
```

sur plusieurs lignes

avec beaucoup d'espaces et des sauts de ligne""")

Ceci est un message

sur plusieurs lignes

avec beaucoup d'espaces et des sauts de ligne

Les opérations d'entrée/sortie (suite)

```
>>> x=10 ; y=15 ; z=28;
```

```
>>> print (x, y, z, sep=' ');
```

```
10 15 28
```

```
>>> print (x, y, z, sep=';');
```

```
10;15;28
```

```
>>> print (x, y, z , sep='\n');
```

```
10
```

```
15
```

```
28
```

```
>>> print ('x =',x,'y =',y, 'z =', z, sep= ' ' , end =';');
```

```
x = 10 y = 15 z = 28;
```

sep désigne le caractère de séparation

end désigne le caractère de marquage de fin

Les opérations d'entrée/sortie (suite)

▮ Opération de lecture : `input()`

```
>>> x=input("saisir : ")
```

Saisir : 3498

```
>>> print("la saisie",x, "est de type",type(x))
```

la saisie 3498 est de type <class 'str'>

Les opérations d'entrée/sortie (suite)

- ❖ Il est toutefois possible de convertir la quantité saisie en entier, réel ou même booléen au moyen de `int`, `float`, et `bool`

```
>>> x=int(input("saisir un entier: "))
```

```
saisir un entier: 12
```

```
>>> print(x, "de type", type(x))
```

```
12 de type <class 'int'>
```

```
# Ou encore en réel
```

```
>>> x=float(input("saisir un réel: "))
```

```
saisir un réel: 23
```

```
>>> print(x, "de type", type(x))
```

```
23.0 de type <class 'float'>
```

Les structures conditionnelles

Attention à l'indentation !!!

if condition1:

instruction 1

elif condition2:

instruction 2

elif condition3:

instruction 3

instruction 4

.

.

.

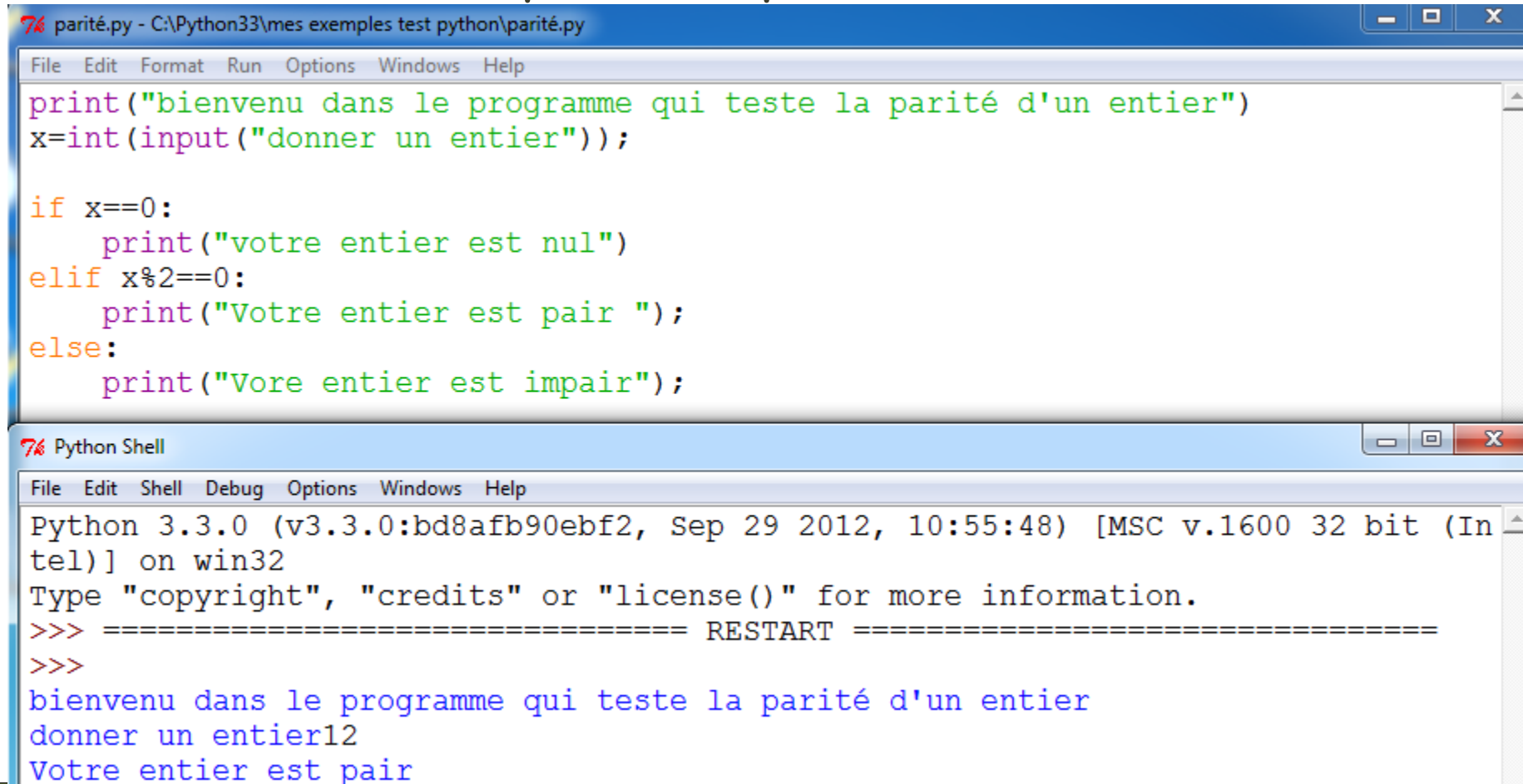
else :

instruction 5

instruction 6

Les structures conditionnelles (suite)

Ecrire un programme qui saisi un nombre et teste si l'entier est nul, pair ou impair



```
7% parité.py - C:\Python33\mes exemples test python\parité.py
File Edit Format Run Options Windows Help
print("bienvenu dans le programme qui teste la parité d'un entier")
x=int(input("donner un entier"));

if x==0:
    print("votre entier est nul")
elif x%2==0:
    print("Votre entier est pair ");
else:
    print("Vore entier est impair");

7% Python Shell
File Edit Shell Debug Options Windows Help
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
bienvenu dans le programme qui teste la parité d'un entier
donner un entier12
Votre entier est pair
```

Les structures itératives : Boucle For

Syntaxe

```
for i in range(a):  
    instructions
```

```
for i in range(a,b):  
    instructions
```

```
for i in range(a,b,c):  
    instructions
```

```
for i in iter:  
    instructions
```

NB: `range(a)` désigne l'intervalle $[0,a[$
`range(a,b)` désigne l'intervalle $[a,b[$
`range(a,b,c)` désigne l'intervalle $[a,b[$ par pas entier égal à c

Le quatrième cas est un parcours par élément que nous pourrons effectuer avec les itérables tel que les listes, les tuples, les chaînes de caractères ou même les fichiers...

Exemple:

```
>>> for i in range(5):  
    if i**2==4:  
        continue  
    print(i)
```

0
1
3
4

```
>>> for i in range(1,11,2):  
    print(i)
```

1
3
5
7
9

```
>>> for i in range(1,5):  
    print(i**2)
```

1
4
9
16

```
>>> L=[1,'bb',-1,"bonjour",(1,2)]
```

```
>>> for elt in L:  
    print(elt)
```

1
bb
-1
bonjour
(1, 2)

L'instruction for ... in

L'instruction **for in iter** : permet d'itérer sur le contenu d'une liste, d'un tuple, les caractères d'une chaîne ou même un fichier ...

```
>>> L=list(range(5))
>>> L
[0,1,2,3,4]
>>> L1=[]
>>> for k in L:
        L1.append(k**2)
>>> L1
[0, 1, 4, 9, 16]

>>> ch="azerty"
>>> ch1=''
>>> for c in ch:
        ch1=ch1+c*2
aazzeerrttyy
```


Construction de listes par compréhension !!

```
>>> L= [i for i in range(1,21,2)]
```

```
>>> L
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

```
>>> L=[(i,j) for i in range(1,5) for j in range(1,5)]
```

```
>>> L
```

```
[(1, 1), (1, 2), (1, 3), (1, 4), (2, 1), (2, 2), (2, 3), (2, 4), (3, 1), (3, 2), (3, 3), (3, 4), (4, 1), (4, 2), (4, 3), (4, 4)]
```

```
>>> L=[(i,j) for i in range(1,5) for j in range(1,5) if (i+j)%2 ==0]
```

```
>>> L
```

```
[(1, 1), (1, 3), (2, 2), (2, 4), (3, 1), (3, 3), (4, 2), (4, 4)]
```

Les structures itératives conditionnelles : Boucle tant que

▮ Syntaxe

```
while condition :  
    instructions
```

▮ Exemple

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

1
2
3
4
5

Les sous programmes

Un programme sous Python se déclare à l'aide de l'instruction `def`

Syntaxe :

```
def nom_programme(liste_paramètres p1,p2...):  
    instructions  
    return(résultat)
```

Exemple :

```
def saisi():  
    """cette fonction permet de saisir un entier >0  
    et retourne cet entier """  
  
    while True :  
        x=int(input("saisir un entier >0: "))  
        if x>0:  
            break  
  
    return(x)
```

Les sous programmes (suite)

```
>>> x=saisi()
```

```
saisir un entier >0: -5
```

```
saisir un entier >0: 45
```

```
>>> help(saisi)
```

```
Help on function saisir in module __main__:
```

```
cette fonction permet de saisir un entier >0  
et retourne cet entier
```

```
saisi()
```

Les sous programmes -exemples

```
def saisi_1():  
    x=int(input("saisir un entier"))  
    return(x)
```

```
def saisi_2():  
    x=int(input("saisir un entier"))  
    y=int(input("saisir un entier"))  
    return(x,y)
```

```
A=saisi_1()  
print(A)  
B,C=saisi_2()  
print(B,C)
```

```
>>> #trace d'exécution  
saisir un entier12  
12  
saisir un entier34  
saisir un entier12  
34 12  
>>> A  
12  
>>> B,C  
(34, 12)
```

Les sous programmes :

Passage de paramètres: exemples

```
def double_3(x):  
    y=x  
    x=2*x  
    print("le double de", y, "est", x)  
    return(x)
```

```
print(A)  
double_3(A)  
print("la valeur de ",A, "reste inchangée")
```

```
    ou encore  
print(A)  
A=double_3(A)  
print("la valeur de ",A, "modifiée")
```

Les sous programmes - Passage de paramètres par valeur: exemples

```
A=saisi()
print(A)
double_3(A)
print("la valeur de ",A, "reste inchangée, le travail a été effectué sur une copie locale")

#ou encore
print(A,"en entrée Mais avec A=double_3(A)")
A=double_3(A)
print("la valeur de ",A, "a été modifiée")
```

```
saisir un entier >012
12
le double de 12 est 24
la valeur de 12 reste inchangée, le travail a été effectué sur une copie locale
12 en entrée Mais avec A=double_3(A)
le double de 12 est 24
la valeur de 24 a été modifiée
```

Les sous programmes - Passage de paramètres: Exemples

```
def double_4(x,y):  
    y=2*x  
    print("le double de", x,"est", y)  
    return(y)  
A=saisi()  
A=double_4(A,B)  
print("la valeur de A= ",A, "modifiée")
```

>>> Saisir un entier > 0: 23

Traceback (most recent call last):

File "C:\Users\Desktop\livre python\Code Python\saisi.py",
line 71, in <module>

A=double_4(A,B)

NameError: name 'B' is not defined

Les sous programmes - Passage de paramètres par référence

Attention ! les types modifiables (tels que les listes) peuvent être modifiés au sein d'une fonction

```
>>> def MaFonction():  
... liste[1] = -127  
...
```

```
>>> def MaFonction(x):  
... x[1] = -15  
...
```

```
>>> liste = [1,2,3]  
>>> MaFonction()  
>>> liste  
[1, -127, 3]
```

```
>>> y = [1,2,3]  
>>> MaFonction(y)  
>>> y  
[1, -15, 3]
```

Les sous programmes - Passage de paramètres par référence

Pour éviter ce problème, on peut utiliser des tuples puisque ces derniers sont non modifiables !

```
>>> def mafonc(x):  
... x[1] = -15  
...  
>>> z=(1,2,2)  
>>> z  
(1, 2, 2)  
>>> mafonc(z)  
Traceback (most recent call last):  
File "<pyshell#24>", line 1, in <module> mafonc(z)  
File "<pyshell#17>", line 2, in mafonc x[1]=-15  
TypeError: 'tuple' object does not support item assignment
```

Les sous programmes - Passage de paramètres par référence

Une autre solution pour éviter la modification d'une liste lorsqu'elle est passée en tant qu'argument,

```
>>> def mafonction(x):  
... x[1] = -15  
...  
>>> y = [1,2,3]
```

```
>>> mafonction(y[:])  
>>> y  
[1, 2, 3]  
>>> mafonction(list(y))  
>>> y  
[1, 2, 3]
```

Les sous programmes - Passage de paramètres par référence : exemple

```
def ajout_elt(L, x):  
    L.append(x)  
  
L=list(range(6))  
print(L)  
ajout_elt(L, 45)  
print(L)
```

>>> #trace d'exécution

[0, 1, 2, 3, 4, 5]

[0, 1, 2, 3, 4, 5, 45]

>>>

Les sous programmes :

Passage de paramètres par défaut

```
def double_5(x=10):  
    y=2*x  
    return(y)
```

```
print(double_5())  
x=saisi()  
print(double_5(x))
```

```
>>>  
passage de paramètres par défaut !!  
20  
saisir un entier >05  
10
```

Les sous programmes : Paramètres positionnels

Python autorise d'appeler la fonction "sans trop respecter l'ordre des arguments" au moment de la déclaration. Il suffit d'utiliser le nom du paramètre au moment de l'appel et de préciser la valeur qu'il prend :

```
>>> def Aff(nom, prenom, age):  
    print('nom=', nom)  
    print('prenom=', prenom)  
    print('age=', age)  
>>> Aff(age=20, prenom='ppp', nom='nnn')  
nom= nnn  
prenom= ppp  
age= 20
```

Les sous programmes :

Nombre de paramètres variable

Python permet de définir une fonction avec un nombre arbitraire de paramètres. Généralement ce paramètre est appelé « **args** » ou « **params** » et il est toujours précédé du caractère « ***** ».

Ceci permet une flexibilité sur le nombre des paramètres au moment de l'appel.

```
>>> def somme(*args):  
    s=0  
    for i in args:  
        s+=i  
    return s
```

L'appel de la fonction est basé sur un « tuple » de valeurs :

```
>>> somme()
```

Variables locales vs variables globales

- ▮ **Variable Locale:** Une variable locale est une variable interne à la procédure et n'est visible qu'à l'intérieur de celle-ci. Les valeurs des variables locales ne sont donc pas communiquées à l'extérieur. Une variable locale n'a plus d'existence à la terminaison du programme.
- ▮ **Variable globale :** toute variable définie à l'extérieur des fonctions, connue et utilisable partout dans le programme où elle est définie, en particulier dans n'importe quelle fonction de ce programme.

Les sous programmes - Passage de paramètres: Exemples

Une variable passée en argument est considérée comme **locale** lorsqu'on arrive dans la fonction

Exemple :

```
>>> def MaFonction(x):  
...   print('x vaut',x,'dans la fonction')  
>>> MaFonction(2)  
x vaut 2 dans la fonction  
>>> print(x)  
Traceback (most recent call last):  
File "<stdin>", line 1, in ?  
NameError: name 'x' is not defined
```

Les sous programmes - Passage de paramètres: Exemples

Lorsqu'une variable déclarée à la racine du programme, elle est considérée **globale**, donc visible dans tout le programme.

Exemple :

```
>>> def MaFonction():  
...     print(x)  
...
```

```
>>> x = 3  
>>> MaFonction()  
3  
>>> print(x)  
3
```

- Dans cet exemple, la variable x est visible dans le module principal et dans la fonction.

Les sous programmes - Passage de paramètres: Exemples

Toutefois, Python ne permet pas la modification d'une variable globale dans une fonction

Exemple :

```
>>> def MaFonction():  
... x = x + 1  
>>> x=1  
>>> MaFonction()
```

Les sous programmes - Passage de paramètres: Exemples

Si on veut vraiment modifier une variable globale dans une fonction, il faut utiliser le mot-clé **global**

Exemple :

```
>>> def MaFonction():  
...   global x  
...   x = x + 1  
...
```

```
>>> x=1  
>>> MaFonction()  
>>> x  
2
```

Dans ce dernier cas, le mot-clé **global** a forcé la variable « x » à être **globale** plutôt que **locale** au sein de la fonction.

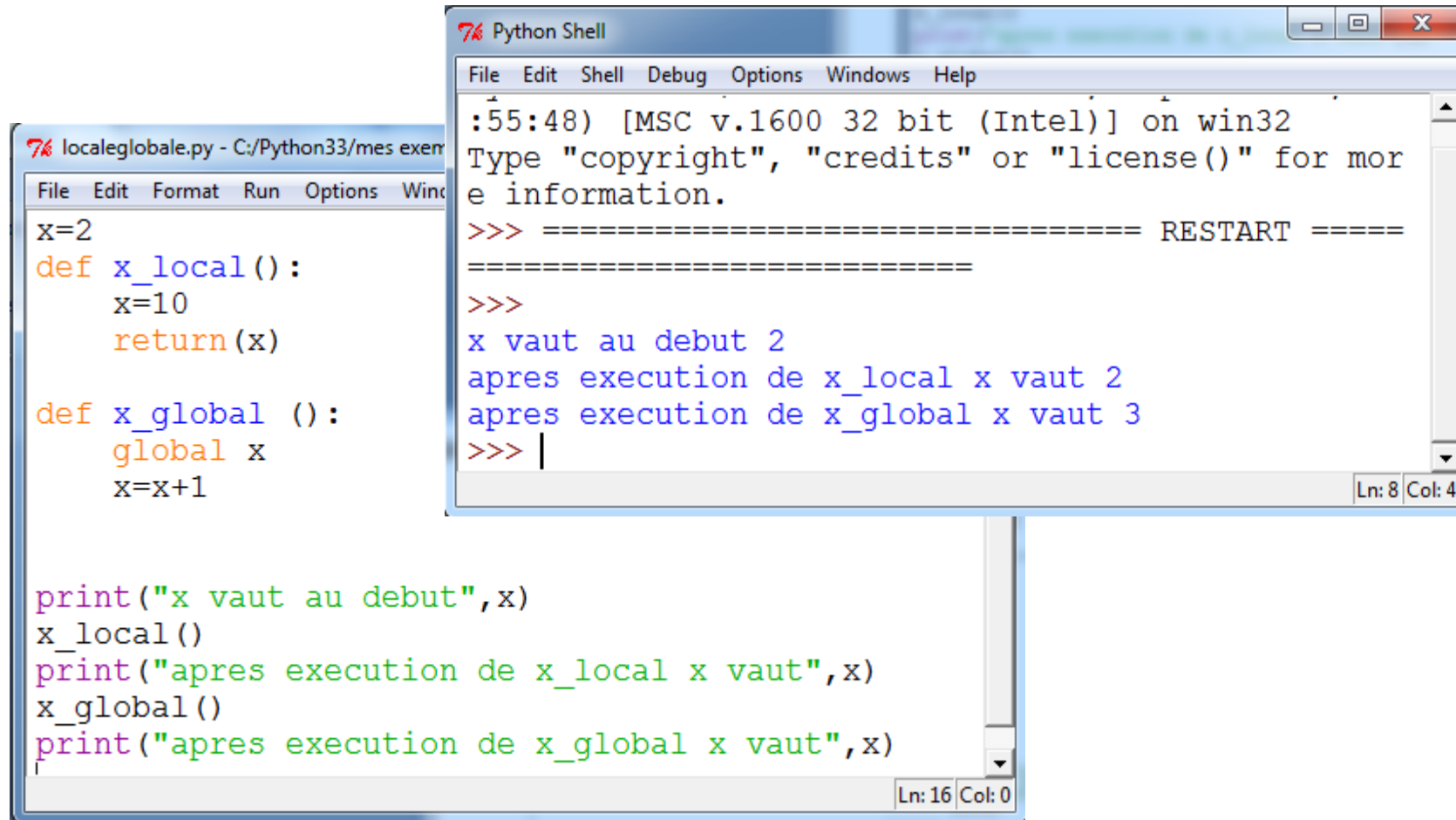
Les sous programmes - Passage de paramètres: Exemples

Application :
Exécuter à la main le programme suivant :

```
def f():  
    global a  
    a = a + 1  
    c = 2 * a  
    return a + b + c
```

```
a=3  
b=4  
c=5  
print(f())  
print(a)  
print(b)  
print(c)
```

Variables locales vs variables globales



The image shows a Python IDE with two windows. The left window, titled 'localeglobale.py', contains the following code:

```
x=2
def x_local():
    x=10
    return(x)

def x_global():
    global x
    x=x+1

print("x vaut au debut",x)
x_local()
print("apres execution de x_local x vaut",x)
x_global()
print("apres execution de x_global x vaut",x)
```

The right window, titled 'Python Shell', shows the execution output:

```
:55:48) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more
>>> ===== RESTART =====
>>>
x vaut au debut 2
apres execution de x_local x vaut 2
apres execution de x_global x vaut 3
>>> |
```

The status bar of the shell window shows 'Ln: 8 Col: 4'.

Fonction lambda

Une fonction **lambda** est une fonction anonyme à laquelle on n'a pas donné de nom et définie par une expression simple.

Une telle fonction s'écrit « **lambda x: e** », où « e » est une expression pouvant comporter la variable « x ». Elle désigne la fonction $x \mapsto e(x)$

La syntaxe est :

lambda paramètres : expression

Exemple: les deux définitions suivantes de la fonction f sont équivalentes :

```
def carre(x):  
    return x*x
```

```
f = lambda x: x*x
```

Fonction lambda

```
>>> f=lambda x:x**2
>>> X=list(range(11))
>>> Y=[]
>>> for e in X:
>>>     Y.append(f(e))
>>> print(X,'\n',Y)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```


Try... except Exemple..

```
def Modif_liste(L):  
    i=len(L)+1  
    x=-1  
    while x<0:  
        x=int(input("Donner la nouvelle valeur : "))  
    while i>len(L):  
        i=int(input("Donner l'indice de l'élément à modifier"))  
        L[i]=x  
  
    return(L)
```

Try... except Exemple..

```
def Modif_liste(L):  
    i=len(L)+1  
    x=-1  
    while x<0:  
        try :  
            x=int(input("Donner la nouvelle valeur : "))  
        except ValueError:  
            print("erreur de saisie, saisissez un entier SVP")  
    while i>len(L):  
        try :  
            i=int(input("Donner l'indice de l'élément à modifier"))  
            L[i]=x  
        except IndexError:  
            print("erreur d'indice un entier compris entre 0 et ", len(L))  
        except ValueError:  
            print("erreur de type")  
  
    return (L)
```

Gestion des erreurs

la clause TRY...EXCEPT

try:

instruction_à_faire

except <<type_exception_optionnel>>

instruction_à_exécuter_si_une_exception_se_déclenche

D'abord, la *clause d'essai* (clause **try** : les instructions entre **try** et **except**) est exécutée.

- ▮ Si pas d'exception, la clause d'exception est ignorée, et l'exécution du **try** est terminée.
- ▮ Si une exception se produit à un moment de l'exécution de la clause d'essai, le reste de la clause **try** est ignoré.

Puis si son type correspond à l'exception donnée après le mot-clé **except**, la clause **except** est exécutée, puis l'exécution reprend après l'instruction **try**.

-Si une exception se produit qui ne correspond pas à l'exception donnée dans la clause **except**, elle est renvoyée aux instructions **try** extérieures; s'il n'y a pas de prise en charge, il s'agit d'une *exception non gérée* et l'exécution est arrêtée avec un message.

Gestion des erreurs la clause TRY...EXCEPT

```
def saisi_controlée():  
    x=-1  
    while x<0:  
        try :  
            x=int(input("donner un entier"))  
            if x>0:  
                break  
        except:  
            print("erreur de saisie, saisissez un entier SVP")  
  
x=saisi_controlée()
```

Plan

Chapitre II : Les conteneurs

- Les listes
- Les chaînes de caractères
- Les tuples, dictionnaires et ensembles

Les conteneurs sous Python : types structurés

Un *conteneur* est un objet composite destiné à contenir d'autres objets: nous distinguons les séquences, les dictionnaires (les tableaux associatifs), les ensembles et les fichiers textuels.

Les *conteneurs* sont des objets itérables.

Deux classements sont possibles :

1) Mutables et Non Mutables

mutable : modification autorisée

non mutable: modification non autorisée

2) Ordonnés et Non ordonnés

Les conteneurs sous Python : Les séquences

Une **séquence** est un conteneur ordonné d'éléments indexés par des entiers indiquant leur position dans le conteneur. Les indices commencent par 0.

3 Types de séquences :

- ▮ Les listes : `['z', 1, 2]`
- ▮ Les chaînes : `"azerty"`
- ▮ Les tuples : `(1,2,3,8)`

Les opérations sur les séquences sont les mêmes.
Nous pouvons réaliser les traitements suivants :

Les conteneurs sous Python : Les séquences (suite)

Si S est une séquence :

- $S[i]$: retourne $(i+1)^{\text{ième}}$ élément de S .
 - $S[:a]$: sous séquence d'éléments de début jusqu'à l'élément d'indice a exclu.
 - $S[a:]$: sous séquence d'éléments de l'indice a inclus jusqu'à la fin.
 - $S[:]$: toute la séquence.
 - $S[a:b]$: sous séquence d'éléments allant de l'indice a inclu jusqu'à l'élément d'indice b exclu.
 - $S[a:b:c]$: sous séquence d'éléments de l'indice a inclu jusqu'à indice b exclu par pas égal à c .
- ▮ $\text{len}(S)$ nombre d'éléments .
- ▮ Un indice négatif permet d'accéder aux éléments à partir de la fin (dernier $S[-1]$ ou $S[\text{len}(L)-1]$, l'avant dernier $S[-2]$, ...).

Les séquences : Les listes

Une liste est une collection ordonnée et modifiable d'objets éventuellement hétérogènes séparés par des virgules et définis entre crochets [].

```
>>> dir (list)
```

```
['__add__', '__class__', '__contains__', '__delattr__',  
 '__delitem__', '__doc__', '__eq__', '__format__', '__ge__',  
 '__getattribute__', '__getitem__', '__gt__', '__hash__',  
 '__iadd__', '__imul__', '__init__', '__iter__', '__le__', '__len__',  
 '__lt__', '__mul__', '__ne__', '__new__', '__reduce__',  
 '__reduce_ex__', '__repr__', '__reversed__', '__rmul__',  
 '__setattr__', '__setitem__', '__sizeof__', '__str__',  
 '__subclasshook__', 'append', 'count', 'extend', 'index', 'insert',  
 'pop', 'remove', 'reverse', 'sort']
```

Les listes : Indexation et Slicing

Création et Accès	Slicing
<pre>>>> L=[1,2, 3,6,1.3, 'hello', "bb"] >>> len(L) 7</pre>	<pre>>>> L[3:] [6, 1.3, 'hello', 'bb']</pre>
<pre>>>> L[0] 1</pre>	<pre>>>> L[:5] [1, 2, 3, 6, 1.3]</pre>
<pre>>>> L[len(L)-1] 'bb'</pre>	<pre>>>> L[2:5] [3, 6, 1.3]</pre>
<pre>>>> L[-1] 'bb'</pre>	<pre>>>> L[1:7:2] # ou L[1::2] [2, 6, 'hello']</pre>
<pre>>>> L[:] [1, 2, 3, 6, 1.3, 'hello', 'bb'] >>> L[7] ...IndexError: list index out of range</pre>	<pre>>>> L[::2] [1, 3, 1.3, 'bb'] >>> L[::-1] ['bb', 'hello', 1.3, 6, 3, 2, 1]</pre>

Les listes (suite)

Création (suite)

```
>>> L_vide=[]
```

```
>>> L=[1,3,5,2]
```

```
>>> L+[0,3,1] # l'opérateur + concatène les listes et donne une nouvelle liste  
[1,3,5,2,0,3,1]
```

```
>>> L # L reste inchangée !!!!  
[1,3,5,2]
```

```
>>> L*2 # L'opération list*n ou n*list concatène n copies de la liste  
[1,3,5,2, 1,3,5,2]
```

```
>>> L=L*2
```

```
>>> L
```

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

```
>>> L=list(range(5))
```

```
>>> L
```

```
[0, 1, 2, 3, 4]
```

```
>>> L+[1]*4
```

```
[0, 1, 2, 3, 4, 1, 1, 1, 1]
```

```
>>> # L → [0, 1, 2, 3, 4]
```

Les listes (suite)

▮ Copie d'une liste

```
>>> L=list(range(10))
```

```
>>> L
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> L1=L # Attention les deux listes référencient le  
même objet en mémoire
```

```
>>> L.append(5) # ajoute 5 en fin de liste
```

```
>>> L1
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 5]
```

```
>>> LL=L1.insert(0,10) # insertion de l'objet 10 en position 0
```

```
>>> L1
```

```
[10,0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 5]
```

```
>>> id(L), id(L1)
```

```
(46713952, 46713952)
```

Liste - copie de liste (suite)

▮ Pour recevoir une copie de liste indépendante, plusieurs manières existent:

```
>>> L = [0, 1, 2, 3, 4]
>>> L1 = L
>>> L2=L[:]
>>> id(L),id(L1),id(L2)
(46713952, 46713952, 37856880)
```

Ou encore

```
>>> from copy import *
>>> L3=copy(L)
>>> id(L), id(L1),id(L2), id(L3),
(46713952, 46713952, 37856880, 46752632)
>>> L is L1
True
>>> L is L2
False
>>> L == L2
True
```

Listes : Méthodes prédéfinies

```
>>> nbr=[17,38,10,25,72]
>>> nbr.sort()
>>> nbr
[10, 17, 25, 38, 72]
# nbr.sort(reverse=True) : ordre décroissant
>>> nbr.append(12)
>>> nbr
[10, 17, 25, 38, 72, 12]
>>> nbr.reverse()
>>> nbr
[12, 72, 38, 25, 17, 10]
>>> nbr.remove(38)
>>> nbr
[12, 72, 25, 17, 10]
>>> nbr.index(17)
3
```

```
>>> nbr[0]=11
>>> nbr
[11, 72, 25, 17, 10]
>>> nbr[1:3]=[14,17,2]
>>> nbr
[11, 14, 17, 2, 17, 10]
>>> nbr.pop()
10
>>> nbr.pop(4)
17
>>> nbr
[11, 14, 17, 2]
>>> nbr.extend([17,3,6])
>>> nbr
[11, 14, 17, 2, 17, 3, 6]
>>> nbr.count(17)
2
>>> del nbr[1] ; nbr
[11, 17, 2, 17, 3, 6]
```

Les séquences :

Les chaînes de caractères

- ▮ Une chaîne de caractère est une séquence de caractère unicode définie par la classe str.
- ▮ C'est une séquence de caractères entre simple ou double côtes **non modifiable** !
- ▮ Même principe d'indexation que les listes

Exemple:

```
>>> ch1='toto'
>>> type(ch1)
<class 'str'>
>>> ch_vide=''
>>> ch_vide1=""
>>> ch_vide==ch_vide1
True
```

Les chaînes de caractères (suite)

```
>>> ch="bonjour"
>>> ch_vide=""
>>> ch_n_vide=" "
>>> ch_vide==ch_n_vide
False
>>> ch[0]
'b'
>>> ch[-1]
'r'
>>> len(ch) 7
>>> ch[:3]
'bon'
```

```
>>> ch[3:]
'jour'
>>> ch[2:5]
'njo'
>>> ch[::-1]
'ruojnob'
```

Mais

```
>>> ch[0]='d'
```

TypeError: 'str' object does not support item assignment

Les chaînes de caractères (suite)

Opérateurs + et *

```
>>> ch='bon'
```

```
>>> ch1='jour'
```

```
>>> ch+ch1
```

```
'bonjour'
```

```
>>> ch2=ch+ch1
```

```
>>> ch2*3
```

```
'bonjourbonjourbonjour'
```

Les chaînes de caractères (suite)

```
>>> dir(str)
['__add__', 'index', 'isalnum', 'isalpha', 'isdecimal',
'isdigit', 'islower', 'isnumeric', 'isspace', 'istitle', 'isupper',
'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition',
'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit',
rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase',
'title', 'translate', 'upper', 'zfill']
```

```
>>> help(str.isalpha)
Help on method_descriptor:
```

```
isalpha(...)
    S.isalpha() -> bool
```

Return True if all characters in S are alphabetic
and there is at least one character in S, False otherwise.

Les chaînes de caractères : Méthodes pour les chaînes

```
>>> ch="TototiTi"
```

▮ `isupper()` et `islower()` : retournent `True` si la chaîne ne contient respectivement que des majuscules/ minuscules :

```
>>> ch.isupper()
```

`False`

▮ `istitle()` : retourne `True` si seule la première lettre de chaque mot de la chaîne est en majuscule :

```
>>> ch.istitle()
```

`False`

▮ `isalnum()`, `isalpha()`, `isdigit()` et `isspace()` : retournent `True` si la chaîne ne contient respectivement que des caractères alphanumériques, alphabétiques, numériques ou des espaces :

```
>>> ch.isalpha()
```

`True`

Les chaînes de caractères : Méthodes pour les chaînes

```
>>> mot.upper() # le résultat est une nouvelle chaîne
'TOTOTITI'
>>> mot
'TotoTiTi'
>>> mot.lower()
'tototiti'
>>> mot=mot.upper()
>>> mot
'TOTOTITI'
Autres Méthodes utiles
>>> ch= ' bonjour tout le monde '
>>> ch.strip()
'bonjour tout le monde'
>>> ch.split() → ['bonjour', 'tout', 'le', 'monde']
```

Les séquences : Les tuples

Un tuple (appelé également n-uplet) est une collection ordonnée et non modifiable d'éléments éventuellement hétérogènes.

Syntaxe

Éléments séparés par des virgules, et entourés de parenthèses.

```
>>> t = 12345, 54321, 'salut!' # t = (12345, 54321, 'salut!')
```

```
>>> t
```

```
(12345, 54321, 'salut!')
```

```
>>> t[0]
```

```
12345
```

Les Tuples peuvent être imbriqués:

```
>>> u = t, (1, 2, 3, 4, 5)
```

```
>>> u
```

```
((12345, 54321, 'salut!'), (1, 2, 3, 4, 5))
```

Les tuples (suite)

Un problème particulier (tuple 0 ou 1 élément):

- ▮ Les tuples vides sont construits avec des parenthèses vides;
- ▮ Un tuple avec un élément est construit en faisant suivre une valeur d'une virgule

Par exemple :

```
>>> tuple1 = ()
```

```
>>> tuple2 = 'salut', ← # notez la virgule en fin de ligne
```

```
>>> len(tuple1 ), len(tuple2 ), tuple2
```

```
(0, 1, ('salut',))
```

Les tuples (suite)

Remarque :

Les chaînes et les Tuples n'étant pas modifiable, il est possible de migrer vers des listes au moyen de l'instruction **list**:

```
>>> t=(1,2,3)
```

```
>>> t=list(t)
```

```
>>> type(t)
```

```
<class 'list'>
```

```
>>> t.append(5)
```

```
>>> t
```

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

```
>>> ch="Bonjour "
```

```
>>> List (ch)
```

```
['b', 'o', 'n', 'j', 'o', 'u', 'r']
```

NB: Une fonction qui retourne plusieurs résultats retourne un tuple !!!!

Les dictionnaires

Un dictionnaire (tableau associatif) est un type de données permettant de stocker des couples **clé:valeur**, avec un accès très rapide à la valeur à partir de la clé, la clé ne pouvant être présente qu'une seule fois dans le tableau.

Caractéristiques d'un dictionnaire :

- ▮ La fonction **len()** donne le nombre de couples stockés ;
- ▮ Permet de retrouver un objet par sa clé
- ▮ Il est *itérable* (on peut le parcourir)
- ▮ Non ordonné
- ▮ Mutable.

Python propose la class **dict**.

Les dictionnaires (suite)

▮ Pour créer un dictionnaire, on utilise la syntaxe :

```
dictionnaire = {cle1:valeur1,cle2:valeur2, cleN:valeurN}
```

```
>>>d_vide={}
```

```
>>>d={1:'un', 2:'Deux',3:'trois',4:'Quatre', 5:'cinq'}
```

```
>>> type(d)
```

```
<class 'dict'>
```

```
>>> dir(dict)
```

```
['__class__', '...', '__repr__', '__setattr__', '__setitem__', '__sizeof__', '__str__',  
 '__subclasshook__', 'clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem',  
 'setdefault', 'update', 'values']
```

```
>>> help(dict.keys)
```

Help on method_descriptor:keys(...)

D.keys() -> a set-like object providing a view on D's keys

▮ On peut ajouter ou remplacer un élément dans un dictionnaire :

```
dictionnaire[cle]= valeur
```

```
>>>d[6]='six' # ajoute un couple clé valeur
```

```
>>>d.__setitem__(7,'sept') # ajoute un couple clé valeur
```

```
>>>d
```

```
{1: 'un', 2: 'Deux', 3: 'trois', 4: 'Quatre', 5: 'cinq', 6: 'six', 7: 'sept'}
```

Les dictionnaires (suite)

- On peut supprimer une clé (et sa valeur correspondante) d'un dictionnaire en utilisant, au choix, le mot-clé `del` ou la méthode `pop`.

```
>>> del d[7]
```

```
>>> d.pop(6)
```

```
'six'
```

```
>>> d
```

```
{1: 'un', 2: 'Deux', 3: 'trois', 4: 'Quatre', 5: 'cinq'}
```

- On peut parcourir un dictionnaire grâce aux méthodes `keys` (parcourt les clés), `values` (parcourt les valeurs) ou `items` (parcourt les couples clé-valeur).

```
>>> coef={"maths":5,"physique":4,"info":4}
```

```
>>> for cle in coef: # ou for cle in coef.keys():  
    print(cle)
```

```
maths
```

```
info
```

```
physique
```

Les dictionnaires (suite)

```
>>>for valeur in coef.values():  
    print(valeur)
```

```
5  
4  
4
```

```
>>>for matiere,coeff in coef.items():  
    print(matiere, ': ', coeff)
```

```
maths : 5  
info : 4  
physique : 4
```

```
>>>coef.items()  
dict_items([('maths', 5), ('info', 4), ('physique', 4)])
```

```
>>>coef.keys()  
dict_keys(['maths', 'info', 'physique'])
```

```
>>> coef.values()  
dict_values([5, 4, 4])
```

Les ensembles

Un ensemble est une collection **non ordonnée** et **mutable** d'objets **uniques**. Les ensembles sont des structures non indexées !

Syntaxe

Ensemble d'objets séparés par des virgules entre {}

Exemple

```
>>> E_vide=set() # crée un ensemble vide
>>> S={1,1,3,3,'a','b','ccc',2,2,1,'a','a','bb'}
>>> S
{'a', 1, 2, 'b', 'bb', 3, 'ccc'}
>>> S.add(9)
>>> S
{'a', 1, 2, 'b', 'bb', 3, 'ccc', 9}
```

Les ensembles (suite) Opérations...

```
>>> e=set({1,2,3,'E','E','R','A',3,3,2,6})
>>> e1=set([1,2,3,'k','j',2,2,'L','A',3,3,2,6])
>>> type(e),type(e1)
<class 'set'> <class 'set'>
>>> e|e1    # opération d'union
{'A', 1, 2, 3, 'E', 6, 'k', 'j', 'L', 'R'}
>>> e&e1    # opération d'intersection
{'A', 1, 2, 3, 6}
>>> e-e1    # opération de différence
{'R', 'E'}
>>> e^e1    # ou exclusif
{'E', 'k', 'j', 'L', 'R'}
```

Plan

Chapitre III : Bibliothèques standards

- Calcul scientifique : Numpy
- Les graphiques : Matplotlib
- Visualisation et analyse de données : Pandas

Bibliothèque Numpy

- ▮ NumPy est une bibliothèque Python très populaire qui est principalement utilisée pour effectuer des calculs mathématiques et scientifiques. Elle offre de nombreuses fonctionnalités et outils qui peuvent être utiles pour les projets de Data Science.
- ▮ Python ne propose de base que le type `list`, conteneur dynamique hétérogène puissant, mais non orienté au calcul numérique !!
- ▮ Le module **numpy** propose un ensemble de classes et de fonctions dédiés aux calculs numériques :
 - ▮ La classe **ndarray** (N dimensional array) : tableaux homogènes multi-dimensionnels
 - ▮ **numpy.random** : un module pour les générateurs aléatoires

Bibliothèque Numpy: La fonction array

```
>>> M=np.ndarray((2,2))    #création de tableau 2x2 de valeurs aléatoires, par défaut réelles mais on peut changer le type à l'aide du parameter dtype !
```

```
>>> print(M)
```

```
[[ 4.24510811e+175, 1.66039110e+243]
 [ 2.59345432e+161, 1.96507163e-147]]
```

▮ La fonction `array` convertit un objet list en objet ndarray :

```
>>> m1 = np.array([1, 2, 3]) ;
```

```
>>> print(m1)
```

```
array([1, 2, 3])
```

▮ La méthode `tolist` fait le travail inverse : elle crée un objet list, copie de l'objet ndarray :

```
>>> m1.tolist()
```

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


Bibliothèque Numpy: autres générateurs de tableaux

▮ Les fonctions `linspace`, `logspace` créent des vecteurs de float (très utile pour le traçage des courbes) :

```
>>> np.linspace(0, 10, 5) # début, fin, nombre de points,  
array([ 0., 2.5, 5., 7.5, 10.]
```

```
>>> np.logspace(1, 2, 4) # 4 points entre 10**1 et 10**2  
array([ 10., 21.5443469, 46.41588834, 100.]
```

```
>>> np.logspace(1, 2, 4, base=2) # 4 points entre 2**1 et 2**2  
array([ 2. , 2.5198421, 3.1748021, 4.]
```

Bibliothèque Numpy: autres générateurs de tableaux

▮ Les fonctions numpy zeros, ones, eye et diag :

```
>>> m1 = np.zeros(4) #print(m1) donne tableau de 0 [ 0., 0., 0., 0.]
```

```
>>> m2 = np.ones(3, bool) ; # print(m2) donne [True True True]
```

```
>>> np.eye(3); # crée une matrice identité
```

```
array([[ 1., 0., 0.], [0., 1., 0.], [ 0., 0., 1.]])
```

```
>>> np.diag([3,2,8]) #crée une matrice diagonale
```

```
array([[3, 0, 0], [0, 2, 0], [0, 0, 8]])
```

Array, slicing

- ▮ Si M est une matrice
 - ▮ $M[:,:]$: une copie intégrale de M avec nouvelle référence
 - ▮ $M[:a,:b]$: sous matrice dont les lignes sont du début jusqu'à $\langle a \rangle$ exclu, et les colonnes du début jusqu'à $\langle b \rangle$ exclu.
 - ▮ $M[:,b]$: colonne d'indice b # ou $(b+1)$ ème colonne
 - ▮ $M[a,:]$: ligne d'indice a # ou $(a+1)$ ème ligne

Array: accès à un élément

▮ Même indexage que les listes !

```
>>> M=np.array([[2,4,5,11],[4,5,0,1]])
```

```
>>> M
```

```
array([[2, 4, 5, 11],  
       [ 4, 5, 0, 1]])
```

```
>>> M[1,1]
```

```
5
```

```
>>> N=np.array([1,3,6,3,4,8])
```

```
>>> N
```

```
array([1, 3, 6, 3, 4, 8])
```

```
>>> N[-1]
```

```
8
```

```
>>> N[2]=5
```

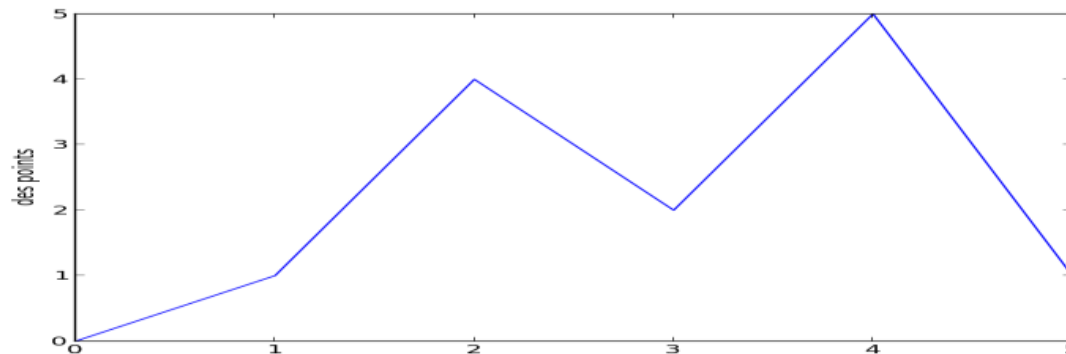
```
>>> N
```

```
array([1, 3, 5, 3, 4, 8])
```

Module Matplotlib

▮ Un module très riche pour le traçage de courbes et de graphiques.

```
import matplotlib.pyplot as plt  
l=[0,1,4,2,5,1]  
plt.plot(l)  
plt.ylabel('des points')  
plt.show()
```



Exemple

Dans l'exemple suivant, nous allons tracer quelques courbes, en variant leurs representations.

▮ Le troisième argument permet de préciser la couleur et le type de trace. Le premier caractère désigne la couleur (**r** pour rouge, **b** pour bleu, **k** pour noir, **g** pour vert, **c** pour cyan, **m** pour magenta, **y** pour jaune, **w** pour blanc).

▮ Les arguments suivants précisent le type du tracé :

- pour des segments de droite, **.** pour des points isolés, **o** pour des « gros points » isolés, **o-** pour des gros points reliés par des segments, etc.

▮ On peut ajouter des arguments optionnels comme par exemple pour préciser la largeur des traits avec `linewidth`.

Graphique

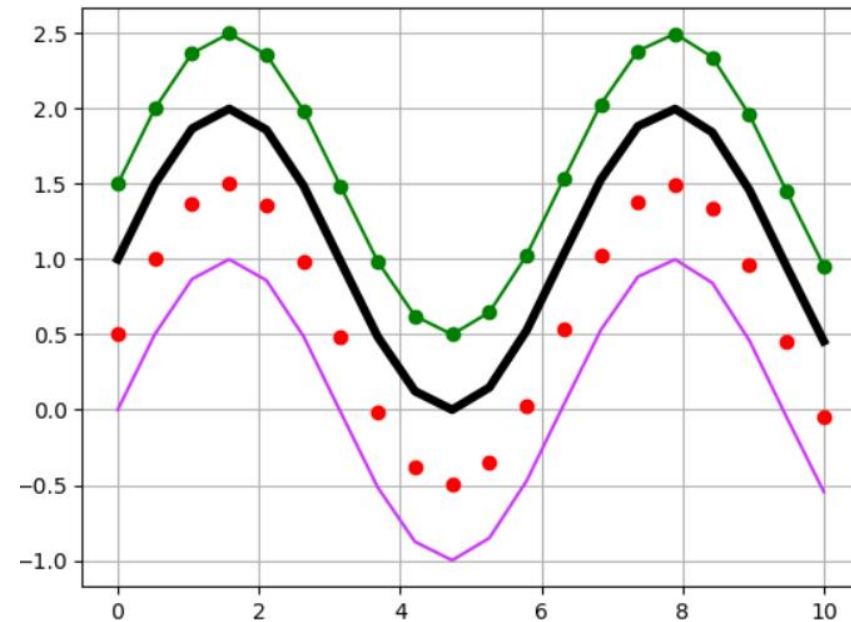
```
from math import *
import numpy as np
import matplotlib as mp
import matplotlib.pyplot as plt
```

```
y = []
t = np.linspace(0.,10.,20)
print(t)
```

```
for k in range(4):
    y.append([k/2.0+sin(x) for x in t])
print(y)
```

```
plt.plot(t,y[0],color =(0.8,0.2,1.0))
plt.plot(t,y[1],'ro')
plt.plot(t,y[2],'k-', linewidth=4)
plt.plot(t,y[3],'go-')
plt.grid()
plt.show()
```

```
[ 0.          0.52631579  1.05263158  1.57894737  2.10526316  2.63157895
 3.15789474  3.68421053  4.21052632  4.73684211  5.26315789  5.78947368
 6.31578947  6.84210526  7.36842105  7.89473684  8.42105263  8.94736842
 9.47368421 10.          ]
[[0.0, 0.5023511546035125, 0.8687296180358695, 0.9999667804441291, 0.860540338374244, 0.4881892088664759, -0.01630136119395508,
-0.5163795995000284, -0.8766880310355383, -0.9997010372394821, -0.8521223683683795, -0.4738975258547864, 0.03259839026803528,
0.5302708154682747, 0.8844134624088179, 0.999169621451985, 0.8434779451109167, 0.45947990361225544, -0.04888675625363447, -0.54
40211108893698], [0.5, 1.0023511546035127, 1.3687296180358695, 1.499966780444129, 1.360540338374244, 0.9881892088664759, 0.4836
986388060449, -0.01637959950002843, -0.3766880310355383, -0.4997010372394821, -0.3521223683683795, 0.026102474145213606, 0.5325
983902680352, 1.0302708154682747, 1.3844134624088178, 1.499169621451985, 1.3434779451109167, 0.9594799036122554, 0.451113243746
3655, -0.044021110889369774], [1.0, 1.5023511546035127, 1.8687296180358695, 1.999966780444129, 1.860540338374244, 1.48818920886
64759, 0.9836986388060449, 0.48362040049997157, 0.12331196896446173, 0.0002989627605178846, 0.1478776316316205, 0.5261024741452
136, 1.0325983902680353, 1.5302708154682747, 1.8844134624088178, 1.999169621451985, 1.8434779451109167, 1.4594799036122554, 0.9
511132437463655, 0.4559788891106302], [1.5, 2.0023511546035127, 2.3687296180358697, 2.499966780444129, 2.360540338374244, 1.988
1892088664759, 1.483698638806045, 0.9836204004999716, 0.6233119689644617, 0.5002989627605179, 0.6478776316316205, 1.02610247414
52136, 1.5325983902680353, 2.030270815468275, 2.384413462408818, 2.499169621451985, 2.3434779451109167, 1.9594799036122554, 1.4
511132437463656, 0.9559788891106302]]
```



Bibliothèque Pandas

- ▮ Pandas : Librairie Python pour extraire, préparer et éventuellement analyser des données
 - ▮ Contient les classes Series et DataFrame (tables de données)
- Elle permet de :
- ▮ lire des fichiers .csv, xls, HTML, XML, JSON, MongoDB, SQL, ...
 - ▮ sélectionner/supprimer/ajouter des lignes et des colonnes, fusionner des DataFrames
 - ▮ gérer les données manquantes et aberrantes
 - ▮ générer des nombres aléatoires
 - ▮ faire des tests statistiques élémentaires
 - ▮ générer des représentations graphiques
 - ▮ etc.

Bibliothèque Pandas

- ▮ Qu'est ce qu'un dataframe ?
 - Tableau de données
 - Colonnes : observations / Lignes : variables observées
 - Attribution de noms aux observations et variables (~ dictionnaire 2D)
- ▮ Pandas :
 - Se restreint à la manipulation, lecture et prétraitement des données.
 - Utilisation d'autres packages, comme Scikit-learn, pour l'analyse de données avancée.

Pandas : Création d'un DataFrame

```
import pandas as pd

data = {'gov': ['gabes', 'tunis', 'sfax', 'sousse', 'beja'], 'loc': ['SudEst', 'NordEst', 'CentreEst', 'CentreEst', 'NordOuest'], 'pop': [450, 2000, 1200, 800, 300]}

df = pd.DataFrame(data, columns=['gov', 'loc', 'pop'])
```

df

	gov	loc	pop
0	gabes	SudEst	450
1	tunis	NordEst	2000
2	sfax	CentreEst	1200
3	sousse	CentreEst	800
4	beja	NordOuest	300

```
df2 = pd.DataFrame(data, columns=['gov', 'loc', 'pop'], index=['a', 'b', 'c', 'd', 'e'])
```

df2

	gov	loc	pop
a	gabes	SudEst	450
b	tunis	NordEst	2000
c	sfax	CentreEst	1200
d	sousse	CentreEst	800
e	beja	NordOuest	300

Pandas : Manipulation d'un DataFrame

```
df['pop']=0
```

```
df
```

	gov	loc	pop
0	gabes	SudEst	0
1	tunis	NordEst	0
2	sfax	CentreEst	0
3	sousse	CentreEst	0
4	beja	NordOuest	0

```
df.at[3,'pop']=100
```

```
df
```

	gov	loc	pop
0	gabes	SudEst	0
1	tunis	NordEst	0
2	sfax	CentreEst	0
3	sousse	CentreEst	100
4	beja	NordOuest	0

Pandas : Lecture de différents formats de données

Lecture d'un fichier csv

```
df = pd.read_csv('temp.csv')
```

df

	Player	Outlook	Temperature	Humidity	Windy	Play golf
0	0	Rainy	Hot	High	False	No
1	1	Rainy	Hot	High	True	No
2	2	Overcast	Hot	High	False	Yes
3	3	Sunny	Mild	High	False	Yes
4	4	Sunny	Cool	Normal	False	Yes
5	5	Sunny	Cool	Normal	True	No
6	6	Overcast	Cool	Normal	True	Yes
7	7	Rainy	Mild	High	False	No
8	8	Rainy	Cool	Normal	False	Yes
9	9	Sunny	Mild	Normal	False	Yes
10	10	Rainy	Mild	Normal	True	Yes
11	11	Overcast	Mild	High	True	Yes
12	12	Overcast	Hot	Normal	False	Yes
13	13	Sunny	Mild	High	True	No

Autres formats (json,xls,hdf5)

```
mydata=pandas.read_json('toto.json')  
mydata=pandas.read_excel('toto.xls')  
mydata=pandas.read_hdf('toto.hdf')
```

Pandas : Manipulation de données

Concaténation de 2 dataframes (axis par défaut = 0 -> concat, lignes)

```
import pandas

d1 = {"Name": ["Pankaj", "Lisa"], "ID": [1, 2]}
d2 = {"Name": "David", "ID": 3}

df1 = pandas.DataFrame(d1, index={1, 2})
df2 = pandas.DataFrame(d2, index={3})

df1
df2

df3 = pandas.concat([df1, df2])

df3
```

	Name	ID
1	Pankaj	1
2	Lisa	2

	Name	ID
3	David	3

	Name	ID
1	Pankaj	1
2	Lisa	2
3	David	3

Concaténation de 2 dataframes (axis = 1 -> concat, colonnes)

```
d1 = {"Name": ["Pankaj", "Lisa"], "ID": [1, 2]}
d2 = {"Role": ["Admin", "Editor"]}

df1 = pandas.DataFrame(d1, index={1, 2})
df2 = pandas.DataFrame(d2, index={1, 2})

df1
df2

df3 = pandas.concat([df1, df2], axis=1)

df3
```

	Name	ID
1	Pankaj	1
2	Lisa	2

	Role
1	Admin
2	Editor

	Name	ID	Role
1	Pankaj	1	Admin
2	Lisa	2	Editor

Pandas : Manipulation de données

Lecture du contenu d'une colonne

Lecture du contenu de deux colonnes

Pandas : Données manquantes et aberrantes

ST_NUM	ST_NAME	NUM_BEDROOMS	OWN_OCCUPIED
104	PUTNAM	3	Y
197	LEXINGTON	3	N
	LEXINGTON	n/a	N
201	BERKELEY	1	12
203	BERKELEY	3	Y
207	BERKELEY	NA	Y
NA	WASHINGTON	2	
213	TREMONT	--	Y
215	TREMONT	na	Y

Pandas : Données manquantes et aberrantes

```
# Affichage de la colonne ST_NUM  
print(df['ST_NUM'])
```

0	104.0
1	197.0
2	NaN
3	201.0
4	203.0
5	207.0
6	NaN
7	213.0
8	215.0

```
# Affichage des valeurs nulles  
print(df['ST_NUM'].isnull())
```

0	False
1	False
2	True
3	False
4	False
5	False
6	True
7	False
8	False

Pandas : Données manquantes et aberrantes

```
# Affichage de la colonne NUM_BEDROOMS  
print(df['NUM_BEDROOMS'])
```

0	3
1	3
2	n/a
3	1
4	3
5	NaN
6	2
7	--
8	na

```
# Affichage des valeurs nulles  
print(df['NUM_BEDROOMS'].isnull())
```

0	False
1	False
2	False
3	False
4	False
5	True
6	False
7	False
8	False

Pandas : Données manquantes et aberrantes

```
# Faire une liste de types de valeurs manquantes  
missing_values = ["n/a", "na", "--"]  
df = pd.read_csv("property data.csv", na_values = missing_values)
```

```
print(df['NUM_BEDROOMS'])
```

0	3.0
1	3.0
2	NaN
3	1.0
4	3.0
5	NaN
6	2.0
7	NaN
8	NaN

```
print(df['NUM_BEDROOMS'].isnull())
```

0	False
1	False
2	True
3	False
4	False
5	True
6	False
7	True
8	True