Le langage Python

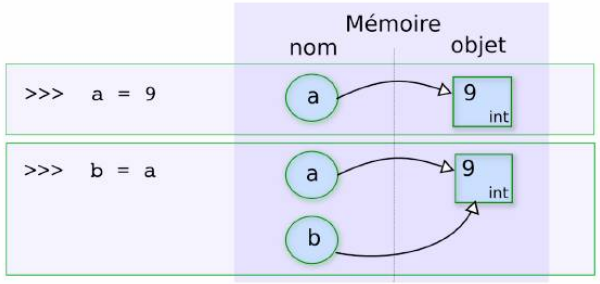
- Le langage Python est constitué :
  - de **mots clefs**, qui correspondent à des instructions élémentaires (`for`, `if`...) ;
  - de **littéraux** : valeurs constantes de types variés (25, `1.e4`, `'abc'`...) ;
  - de **types intrinsèques** (`int`, `float`, `list`, `str`...) ;
  - d'**opérateurs** (=, +, *, /, %...) ;
  - de **fonctions intrinsèques** (*Built-in Functions*) qui complètent le langage.
- L'utilisateur peut créer :
  - des **classes** : nouveaux types qui s'ajoutent aux types intrinsèques ;
  - des **objets** : entiers, flottants, chaînes, fonctions, programmes, modules... instances de classes définies par l'utilisateur ;
  - des **expressions** combinant identificateurs, opérateurs, fonctions...

Python est un langage Orienté Objet

- Avec Python, tout est **objet** : données, fonctions, modules...
- Un **objet** :
  - possède une **identité** (≈ adresse mémoire) ;
  - possède un **type** : un objet est l'instanciation d'une **classe** qui définit son type (type intrinsèque : `int`, `float`, `str`... ou type utilisateur : `class xxx`) ;
  - contient des **données** (exemple : objet numérique -> sa **valeur**).
- Un objet est **référencé** par un **identificateur** :
  - **identificateur**, **référence**, **nom**, **étiquette**... sont des termes synonymes ;
  - **objet** et **variable** sont des termes synonymes (on peut préférer *objet*) ;
  - les noms au format `__xxx__` ont une signification spéciale pour l'interpréteur Python.

Affectation : référence ⇝ objet

- L'affectation opère de droite à gauche :
  - le terme de droite est une **expression**, qui est **évaluée** en tant qu'objet ;
  - le terme de gauche est l'**identificateur** (**référence**, **nom**, **étiquette**) affecté à l'objet évalué.

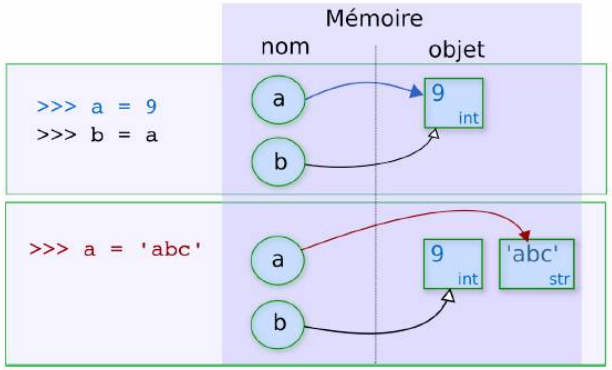


C'est l'objet qui porte le **type** et les **données** (la **valeur**, pour un objet numérique).

- Un objet peut avoir plusieurs noms (*alias*).
- Un objet ne peut pas changer d'identité (≈ adresse mémoire), ni de type.

Affectation : référence ⇝ objet

- Un identificateur associé d'abord à un objet peut ensuite référencer un nouvel objet !



- Quand un objet n'a plus de nom (nombre de références nul), il est détruit (mécanisme automatique de "ramasse-miettes", *garbage collector*).

Objets

- L'affectation = permet d'affecter un **nom** à un objet **créé** ou **existant**.
- `type(•)` renvoie le type de l'objet •.
- `id(•)` renvoie l'identité de l'objet •.
- L'opérateur == teste l'égalité des **valeurs** de 2 objets.
- L'opérateur `is` compare l'**identité** de 2 objets.

```
>>> a = 999  # objet 999, int, nommé a
>>> type(a)
<class 'int'>
>>> b = a   # 2eme nom pour l'objt 999
>>> id(a), id(b)
(33724152, 33724152)
>>> type(b)
<class 'int'>
```

```
>>> c = 999.
>>> id(a), id(b),id(c)
(33724152, 33724152, 38603664)
>>> a == c     # égalité des valeurs ?
True
>>> a is c     # même identité ?
False
>>> a is b     # même identité ?
True
```

Objets

- Le type d'un objet détermine :
  - ▷ les valeurs possibles de l'objet ;
  - ▷ les opérations que l'objet supporte ;
  - ▷ la signification des opérations supportées (opérations *polymorphes*).

```
>>> a = 2 + 3  # addition d'entiers
>>> a
5
>>> 2*a, 2.*a  # mult. dans N, R
(10, 10.0)
>>> 2/3, 2./3. # divis. dans N, R
(0.6666666666666666, 0.666666666...)
```

```
>>>[1, 2]+[5, 6] # concat. des listes
[1, 2, 5, 6]
>>> 2*[5, 6]     # n concaténations
[5, 6, 5, 6]
>>> [5, 6]/2     # pas défini !
...unsupported operand type(s) for /
'list' and 'int'
```

- Un objet dont le contenu peut être changé est dit *mutable* (*non mutable* sinon).

```
>>> a = "Un str est non-mutable !"
>>> a[0]
'U'
>>> a[0] = "u"
...TypeError: 'str' object does
  not support item assignment
```

```
>>> a = [1, 2, 3]  # liste: mutable
>>> a[1]
2
>>> a[1] = 2.3
>>> a
[1, 2.3, 3]
```

Classes et Objets

- `dir(•)` permet d'afficher "ce qu'il y a dans" • (classe, objet, module...).
- On peut utiliser toutes les méthodes (publiques) d'une classe sur un objet instance de la classe : `objet.methode(...)`

```
>>> L1 = [4, 2, 1, 3] # L1 est un objet liste (instance de la classe liste)
>>> type(L1)
<class 'list'>
>>> dir(L1)
['__add__',..., 'append', 'clear', 'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
>>> dir(list)
['__add__',..., 'append', 'clear', 'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
>>> L1.sort() ; L1        # objet.méthode_de_la_classe
[1, 2, 3, 4]
>>> L1.append(5) ; L1     # objet.méthode_de_la_classe
[1, 2, 3, 4, 5]
>>> L1.reverse() ; L1     # objet.méthode_de_la_classe
[5, 4, 3, 2, 1]
```

Types intrinsèques (*built-in types*)

- Type `NoneType` : la seule valeur possible est `None` (≈ objet nul ou vide).

```
>>> a = None
```

  ▷ c'est la valeur retournée par une fonction qui "ne renvoie rien".

- Type `bool` (booléen) : les valeurs possibles sont `True` ou `False` (1, 0).

```
>>> b = True
```

  ▷ la valeur logique de `None` est `False`.

- Types numériques : `int`, `float`, `complex`...

```
>>> c = 123        # int : entier 32 bits, dans [-2147483648, 2147483647]
>>> c = 2**100
1267650600228229401496703205376  # entier en 'précision arbitraire'
>>> d = 1.47e-7    # float : flottants IEEE754 64 bits,
                   #   ~16 chiffres significatifs
                   #   ~10**-308 < valeur absolue < ~10**308
>>> e = 2.1 +3.45j # complex
>>> e.real, e.imag
(2.1, 3.45)
```

Types intrinsèques (*built-in types*)

- Types `list`, `tuple`, `str` : les **séquences** (collections ordonnées) :

```
>>> f = [1.e4, 2e4, 0, 1, e1]
>>> g = (1, 2, 3, e1, f)
>>> s1 = 'l"impact'
>>> s2 = "l'impact"
>>> s3 = "Chaîne accentuée"
```

  - Objet `list` entre `[...]`
  - Objet `tuple` entre `(...)`
  - Objet `str` entre `'...'` ou `"..."`

- Type `dict` (dictionnaire) : **collection non ordonnée** de **paires** (`clef, objet`)

```
>>> d1 = {"Lundi":1, "Mardi":2}
>>> d2 = {"mean":1.2, "stdDev":9.4}
```

  - Objet `dict` : paires `key:value` entre `{...}`

- Type `set` (ensemble) : **collection non ordonnée d'items uniques**

```
>>> s = set([1,5,12,6,"a"])
>>> s = {1,5,12,6,"a"}
```

  - Un `set` est créé avec : `set(...)` ou avec les éléments entre `{...}`

Types intrinsèques conteneurs

- Collections ordonnées : les séquences (*sequences*)
  - les listes `list`
  - les tuples `tuple`
  - les chaînes de caractères `str`
- Collections sans ordre
  - les dictionnaires `dict`
  - les ensembles `set`

Tous les conteneurs Python sont des objets **itérables** : on peut les parcourir avec une boucle `for`.

Séquences : la Classe `list`

- Collection **ordonnée** d'**objets quelconques** (conteneur hétérogène).
- Une liste n'est pas un vecteur ($\rightsquigarrow$ classe **ndarray** du module **numpy**).
- `len(•)` renvoie le nombre d'éléments de la liste •.
- **Indexation** : `•[i]` renvoie l'élément de rang `i` de la liste •
  - ▷ le premier élément de la liste • a le rang 0, le dernier le rang `len(•)-1`.

```
>>> L1 = [10,11,12,13]
>>> L1.append(14)
>>> L1
[10, 11, 12, 13, 14]
>>> L1[0]
10
>>> L1[-1]
14
>>> len(L1)
5
```

```
>>> L1[4]
14
>>> L1[5]
...IndexError: list index out of range
>>> L1[-4]
11
>>> L1[-5]
10
>>> L1[-6]
...IndexError: list index out of range
```

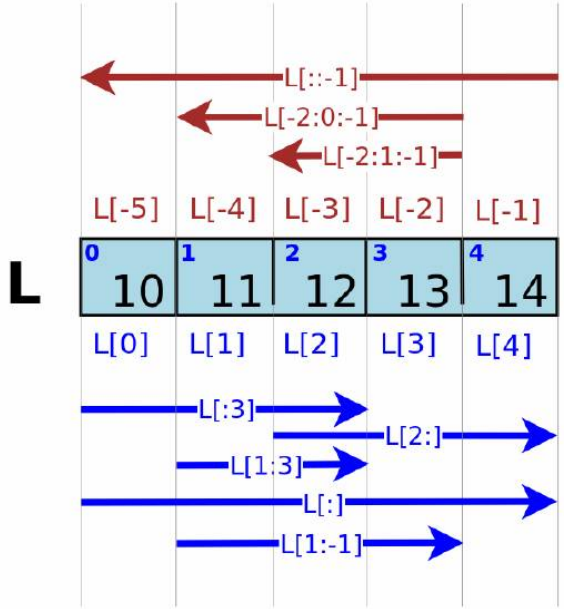
Séquences : la Classe `list`

- **Sous-liste** (*Slicing*) :
  - `•[i:j]` $\rightsquigarrow$ sous-liste de `i` inclus à `j` **exclu**
  - `•[i:j:k]` $\rightsquigarrow$ sous-liste de `i` inclus à `j` **exclu**, par pas de `k`
- Indexation positive, **négative** :
  - $0 \leqslant$ `i` $\leqslant$ `len(•)-1` $\rightsquigarrow$ rang dans la liste
  - `-len(•)` $\leqslant$ `i` $\leqslant$ `-1` $\rightsquigarrow$ `-1` : le dernier, `-2` : l'avant dernier...

```
>>> L1 = [10,11,12,13,14]
>>> L1[1:3]
[11, 12]
>>> L1[:3]
[10, 11, 12]
>>> L1[1:]
[11, 12, 13, 14]
>>> L1[:] # toutes les valeurs de L1
[10, 11, 12, 13, 14]
```

```
>>> L1 = [10,11,12,13,14]
>>> L1[::2]
[10, 12, 14]
>>> L1[::-1]
[14, 13, 12, 11, 10]
>>> L1[-2:1]
[]
>>> L1[-2:1:-1]
[13, 12]
```

Séquences : la Classe list



Séquences : la Classe list

- Une liste est **itérable**, **on peut la parcourir avec une boucle** `for` :

```
>>> L1 = [1, 2, 3]
>>> for a in L1:
    print(a+2)

3
4
5
```

```
>>> col =["red", "green", "blue"]
>>> for i, c in enumerate(col):
    print(i, "Color: " + c)

0 Color: red
1 Color: green
2 Color: blue
```

- L'opérateur + concatène les listes :

```
>>> L1 = [1, 2, 3]
>>> L2 = [4, "a", 5]
>>> L1 + L2
[1, 2, 3, 4, 'a', 5]
```

- L'opération `list*n` ou `n*list` concatène n copies de la liste :

```
>>> [0]*10
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Séquences : la Classe list

- La classe `list` possède des méthodes utiles
  (*cf* `dir(list)` et `help(list.xxxxx)`...)

```
>>> help(list.remove)
Help on method_descriptor:
remove(...)
    L.remove(value) -> None -- remove first occurrence of value.
    Raises ValueError if the value is not present.
```

```
>>> L1 = [10, 11, 12 , 13, 14]
>>> del L1[2]
>>> L1
[10, 11, 13, 14]
>>> L1.remove(11)
>>> L1
[10, 13, 14]
>>> L2 = [10, 11, 10, 13, 10]
>>> L2.count(10)
3
```

```
>>> L2.remove(10)
>>> L2
[11, 10, 13, 10]
>>> L2.remove(10)
>>> L2
[11, 13, 10]
>>> L2.remove(10)
>>> L2
[11, 13]
>>> L2.remove(10)
... list.remove(x): x not in list
```

Séquences : la Classe tuple

- Un tuple est une liste **non mutable** :
  - ▷ ses **éléments** ne supportent pas la **ré-affectation** ;
  - ▷ mais ses **éléments mutables** peuvent être **modifiés**.

```
>>> a  = ()      # tuple vide
>>> b  = (2,)    # 1-tuple
>>> c  = 2,      # 1-tuple
>>> d  = 2, 3, 4 # tuple implicite
>>> d
(2, 3, 4)
>>> t=(1,"non",[1,2,3])
>>> t
(1, 'non', [1, 2, 3])
>>> t[0]=2 # ré-affectation élément
...TypeError: 'tuple' object does
not support item assignment
```

```
>>> t[1]="oui"  # ré-affection élément!
...TypeError: 'tuple' object does not
support item assignment
>>> t[1][0]="N" # élément str non mutable
...TypeError: 'str' object does not
support item assignment
>>> t[2]=[3,4,5] # ré-affection élément
...TypeError: 'tuple' object does not
support item assignment
>>> t[2][0]=-3 # élément list mutable
>>> t
(1, 'non', [-3, 2, 3])
```

Séquences : la Classe `str`

- Chaîne de caractères = **liste** de caractères...
- Même principe d'indexation que les objets `list` :

```
>>>  s = "abcdef"
>>> s[0], s[-1]
('a', 'f')
>>> s[:]
'abcdef'
>>> s[1:-1], s[1:-1:2]
('bcde', 'bd')
```

- De nombreuses méthodes utiles sont proposées par la classe `str` :

```
>>> dir(str)
[... 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith',
'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum',
'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric',
'isprintable', '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']
```

Collection non-ordonnée : la Classe `dict`

- Collection sans ordre de paires `clef`, `objet` : sert à retrouver un objet par sa clef

```
>>> cyl1 = {"L": 1.2, "D": 0.5, "unit": "m"} ; cyl1
{'L': 1.2, 'unit': 'm', 'D': 0.5}
>>> cyl1["L"]
1.2
>>> cyl1["D"] = 0.6 ; cyl1
{'D': 0.6, 'L': 1.2, 'unit': 'm'}
```

- La classe `dict` possède plusieurs méthodes utiles (`dir(dict)` et `help(dict.xxx)`...

```
>>> cyl1["d"] = 0.1 ; cyl1    # création de la clef !
{'d':0.1, 'L': 1.2, 'unit': 'm', 'D': 0.6}
>>> "mass" in cyl1, "D" in cyl1
(False, True)
>>> del cyl1["d"]
>>> list(cyl1.items())
[('L', 1.2), ('unit', 'm'), ('D', 0.6)]
>>> list(cyl1.keys())
['L', 'unit', 'D']
>>> list(cyl1.values())
[1.2, 'm', 0.6]
```

Mots clefs du langage

Doc Python > Language Reference > Identifiers and keywords

docs.python.org/reference/lexical_analysis.html#keywords

Il n'y a que 33 mots clefs (*key words*) dans le langage Python 3.

2.3.1. Keywords

The following identifiers are used as reserved words, or *keywords* of the language, and cannot be used as ordinary identifiers. They must be spelled exactly as written here:

```
False      class      finally    is         return
None       continue   for        lambda     try
True       def        from       nonlocal   while
and        del        global     not        with
as         elif       if         or         yield
assert     else       import     pass
break      except     in         raise
```

Mots clefs du langage

Boucles		Définitions d'objets	
`for`	boucle	`def`	définition d'une fonction
`while`	boucle	`return`	fin fonction, renvoyer valeur
`continue`	tour suivant	`class`	définition d'une classe
`break`	sortie de boucle	`nonlocal`	change la portée d'une variable
		`global`	définit une variable globale

Tests		Opérateurs logiques	
`if`	test	`and`	ET logique
`else`	alternative	`or`	OU logique
`elif`	test combiné	`not`	négation

Mots clefs du langage - suite -

importation d'un **Module**

`import`	module entier
`from`	objets d'un module

Objets

`del`	détruire une référence sur un objet
`in`	parcourir
`is`	comparer

gestion des **Exceptions**

`assert`	définir une assertion
`raise`	créer une exception
`try`	traitement exception
`except`	traitement exception
`finally`	traitement exception

Autre

`pass`	ne rien faire...
`with`	objet dans un context
`as`	utilisé avec `with`, `import`...
`yield`	fonction générateur
`lambda`	fonction *inline*

Les opérateurs

docs.python.org/reference/lexical_analysis.html#operators

Principaux opérateurs

Opérateurs arithmétiques	
`+`	addition
`-`	soustraction
`*`	mutiplication
`/`	division (Python3 : toujours division flottante)
`//`	quotient de la division entière
`**`	exponentiation (0^0 donne 1)
`%`	modulo
`<< n, >> n`	décalage à gauche, à droite de n bits

Principaux opérateurs

Opérateurs de comparaison	
`<, <=`	inférieur strict à, inférieur ou égal à
`>, >=`	supérieur strict à, supérieur ou égal à
`==`	égal à
`!=`	différent de
Opérateurs logiques	
`and`	ET
`not`	négation
`or`	OU
`&`	ET bit à bit
`^`	XOR bit à bit
`|`	OR bit à bit
`~`	complément à 2

Principaux opérateurs

Autres Opérateurs	
`is`	même identité ?
`is not`	identité différente ?
`in`	appartient à ?
`not in`	n'appartient pas à ?
Opérateurs avec affectation	**équivalence**
`x += y`	`x = x + y`
`x -= y`	`x = x - y`
`x *= y`	`x = x * y`
`x /= y`	`x = x / y`
`x //= y`	`x = x // y`
`x **= y`	`x = x ** y`
`x %= y`	`x = x % y`

Fonctions intrinsèques (extraits)

- `abs` valeur absolue, module...
- `bin` convertir en base 2...
- `bool` convertir en booléen...
- `enumerate` parcourir un itérable en numérotant les éléments...
- `eval` interpréter en Python une chaîne de caractère...
- `float` convertir en float...
- `int` convertir en entier...
- `hex` convertir en base 16...
- `max` maximum d'un itérable ou de plusieurs objets...
- `map` appliquer une fonction aux éléments d'une liste...
- `min` minimum d'un itérable ou de plusieurs objets...
- `range` générer une suite de valeurs entières...
- `repr` créer la conversion d'un objet en chaîne de caractères...
- `reversed` parcourir un itérable en sens inverse...
- `str` convertir simplement en chaîne de caractères...
- `sum` somme d'un itérable ou de plusieurs objets...
- `zip` parcourir plusieurs itérables en même temps...

Fonctions intrinsèques (extraits)

- `abs` : valeur absolue ou module

```
>>> abs(-6), abs(-6.7), abs(1+1j)
(6, 6.7, 1.4142135623730951)
```

- `bin`, `hex` : convertit un entier en base 2 ou 16

```
>>> bin(75), hex(64)
('0b1001011', '0x40')
```

- `bool`, `int`, `float` : convertit un objet en type `bool`, `int` ou `float`

```
>>> bool(0), bool(1), bool([]), bool([1]), bool([0]), bool("")
(False, True, False, True, True, False)
>>> int("5"), float("5"), float("1.1e2")
(5, 5.0, 110.0)
```

- `int` : convertit un nombre d'une base B (défaut : 10) en base 10

```
>>> int(2.34), int("45")
(2, 45)
>>> int("110011",2), int("a1",16)
(51, 161)
```

Fonction intrinsèques (extraits)

- `enumerate` : parcourt un itérable en numérotant les éléments

```
>>> L = ["y", "yes", "o", "oui"]
>>> for i, rep in  enumerate(L):
        print((i,rep), end=" ; ")
(0, 'y') ; (1, 'yes') ; (2, 'o') ; (3, 'oui') ;
```

- `eval` : renvoie l'évaluation d'une expression contenue dans une chaîne

```
>>> x = 4
>>> rep = input("Expression en x: ")
Expression en x: x**2 - 6
>>> rep
'x**2-6'
>>> eval(rep)
10
```

- Les fonctions `min`, `max` et `sum` acceptent une liste (un itérable) en argument

```
>>> L1 = [10, 11, 12, 13, 14]
>>> sum(L1)
60
>>> min(L1), max(L1)
(10, 14)
```

```
>>> s="abcde"
>>> sum(s)
TypeError: ...
>>> min(s), max(s)
('a', 'e')
```

Fonctions intrinsèques (extraits)

- `map` : applique une fonction aux éléments d'une liste (renvoie un objet `map` itérable)

```
>>> L = ["1e2", "2.3e-2", "1", "2.3"]
>>> a = map(float, L); a, type(a)
(<map object at 0x7ff5d3dff6a0>, <class 'map'>)
>>> list(a)
[100.0, 0.023, 1.0, 2.3]
```

- `range` (Python 3) : **classe** représentant une séquence d'entiers (prog. arithmétique)

```
>>> a = range(10); a, type(a)
(range(0, 10), <class 'range'>)
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
>>> list(range(0, -5, -1))
[0, -1, -2, -3, -4]
```

très utilisée pour construire les boucles classiques :

```
>>> for i in range(10):
        print(i, end=" ")
0 1 2 3 4 5 6 7 8 9
```

- `reversed` : construit un objet `iterator` représentant une liste en sens inverse

```
>>> L = [1, 2, 3, 4, 5]; list(reversed(L))
[5 4 3 2 1]
```

Définition et appel des fonctions

- **Définition** : mot clef `def` ... terminé par le caractère ':'
  - ▷ le corps de la fonction est **indenté** d'un niveau (bloc indenté) ;
  - ▷ les objets définis dans le corps de la fonction sont locaux à la fonction ;
  - ▷ une fonction peut renvoyer des objets avec le mot clef `return`.
- **Appel** : nom de la fonction suivi des parenthèses `(...)`

```
>>> q = 0
>>> def divide(a,b):
    q = a // b
    r = a - q*b
    return q,r

>>> x, y = divide(5,2)
>>> q, x, y
(0, 2, 1)
```

- La variable q est définie en dehors de la fonction (valeur = 0)
- L'opérateur `//` effectue une division entière
- La variable q définie dans la fonction est locale !

Définition et appel des fonctions

- **Passage des arguments par référence**
  - ▷ à l'appel de la fonction, les arguments (objets) sont transmis par référence ;
  - ▷ un objet mutable transmis en argument peut être modifié par la fonction ;
  - ▷ un objet non mutable transmis en argument ne peut pas être modifié par la fonction.

```
>>> def f0(a, b):
    a = 2
    b = 3
    print("f0-> a: {}, b: {}".format(a,b))
    return
>>> a=0; b=1
>>> a, b
(0, 1)
>>> f0(a, b)
f0-> a: 2, b: 3
>>> a, b
(0, 1)
```

```
>>> def f1(a):
    for i,e in enumerate(a):
        a[i] = 2.*e
    return
>>> a=[1,2,3]
>>> id(a)
140693918338312
>>> f1(a)
>>> id(a)
140693918338312
>>> print(a)
[2.0, 4.0, 6.0]
```

Les Modules

- Un **module** est un **fichier Python**, contenant constantes, fonctions, classes...
- Un module est importé avec `import nomModule`
  - ▷ création d'un **espace de nom** (*namespace*) qui contiendra (préfixe) tous les objets contenus dans le module
  - ▷ **toutes les définitions contenues dans le module sont excécutées**
  - ▷ un identifiant (`nomModule`) est associé à l'espace de nom du module, dans le programme apellant.

Fichier `test.py` :

```
a = 12
def bonjour():
    print("Hello World!")
    return
```

Import du module `test` :

```
>>> import test
>>> x = test.a; print(x)
12
>>> test.bonjour()
Hello World!
```

Les Modules

- Variations possibles pour importer tout ou partie d'un module :
  - ▷ Module **entier** importé avec son espace de nom (*alias* possible) :
    `import module [as alias]`
  - ▷ Module **entier** importé, dans l'espace de nom courant :
    `from module import *`
  - ▷ **Sélection d'items** du module, importés dans l'espace de nom courant :
    `from module import item1 [as alias], item2 [as alias]...`

```
>>> import math
>>> math.sin(math.pi/4)
0.7071067811865475
>>> import math as m
>>> m.sin(m.pi/4)
0.7071067811865475
```

```
>>> from math import *
>>> pi
3.141592653589793
>>> sin(pi/4)
0.7071067811865475
>>> log(1)
0.0
```

```
>>> from math import pi,sin,log
>>> pi
3.141592653589793
>>> sin(pi/4)
0.7071067811865475
>>> log(1)
0.0
```

Module standard **math**

- Donne accès aux constantes et fonctions mathématiques usuelles (trigonométriques, hyperboliques, log, exp...)

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

Module standard **cmath**

```
>>> import math
>>> dir(cmath)
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh',
'atan', 'atanh', 'cos', 'cosh', 'e', 'exp', 'isinf', 'isnan', 'log',
'log10', 'phase', 'pi', 'polar', 'rect', 'sin', 'sinh', 'sqrt', 'tan',
'tanh']
>>> from cmath import exp as cexp
```

Module standard **os**

- Propose des fonctions pour interagir avec le système d'exploitation.
  Exemple : manipulation des répertoires et des fichiers

```
>>> import os
>>> dir(os)
[... 'chdir',... 'getcwd',... 'listdir',... 'mkdir',... 'rmdir']
```

▷ `chdir` (*change directory*) : change le répertoire de travail
▷ `getcwd` (*get current directory*) : renvoit le répertoire de travail
▷ `listdir` (*list directory*) : liste le contenu d'un répertoire (`'.'` : le répertoire courant).

```
>>> os.getcwd()
'/home/jlc'
>>> os.chdir('/tmp/data')
>>> rep = os.getcwd(); rep
'/tmp/data'
>>> os.listdir(rep)
['f1.txt', 'f2.dat', 'f1.dat']
>>> 'f3.dat' in os.listdir(rep)
False
>>> 'f1.dat' in os.listdir(rep)
True
```

```
>>> os.getcwd()
'C:\\Users\\jlc'
>>> os.chdir('C:/tmp/data')
>>> rep = os.getcwd(); rep
'C:\\temp\\data'
>>> os.listdir(rep)
['f1.dat', 'f2.dat', 'f2.txt']
>>> 'f3.dat' in os.listdir(rep)
False
>>> 'f1.dat' in os.listdir(rep)
True
```

Les Entrées/Sorties clavier, écran (*Input/Output*)

- Les Entrée/Sortie clavier/écran permettent le dialogue programme/utilisateur.
- La fonction `input('message')` est utilisée :
  ▷ pour afficher un message à l'écran ;
  ▷ pour **capturer** la **saisie clavier** et la renvoyer comme un `str`.

```
>>> rep = input("Entrer un nombre: ")
Entrer un nombre : 47
>>> rep
'47'
>>> rep == 47
False
>>> type(rep)
<class 'str'>
>>> x = float(rep); x
47.0
>>> type(x); x == 47
<class 'float'>
True
```

- `input` renvoit un `str`
- `float` permet de convertir un `str` en nombre flottant.

Module matplotlib

Tracé d'une courbe y=f(x)

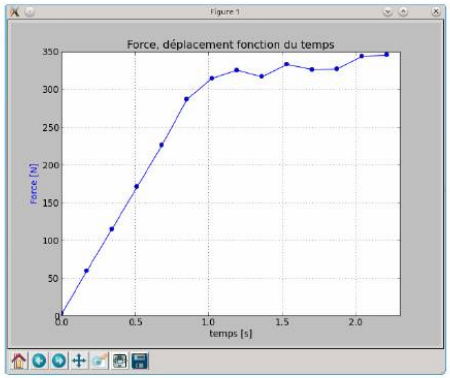
[Plot_XY_21.py]

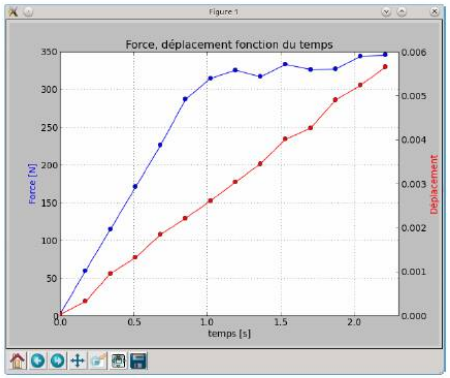
3 colonnes dans le fichier `data1.txt` : temps (s), force (N), déplacement (m).
Le tracé "force=f(temps)" dans le premier système d'axe est fait par :

```
import numpy as np
import matplotlib.pyplot as plt

t, f, d = np.loadtxt("data1Full.txt", unpack = True)
plt.figure()      # Créer une nouvelle figure
plt.title("Force, déplacement fonction du temps")
plt.grid(True)
plt.xlabel("temps [s]")
plt.xlim(0,2.3)
plt.ylabel("Force [N]", color="b")
plt.ylim(0,350.0)
plt.plot(t, f,"o-b")
# copie du tracé dans un fichier :
plt.savefig("data1.png", format = "png")
plt.show()
```

La figure est sur la diapo suivante...

Tracé d'une courbe y=f(x)





Le tracé complet ([Plot_XY_2.py], à droite) est fait en rajoutant les lignes :

```
plt.twinx()    # second système d'axes :
plt.xlim(0, 2.3)
plt.ylim(0, 6.e-3)
plt.ylabel("Déplacement", color="r")
plt.plot(t, d, "o-r")
```

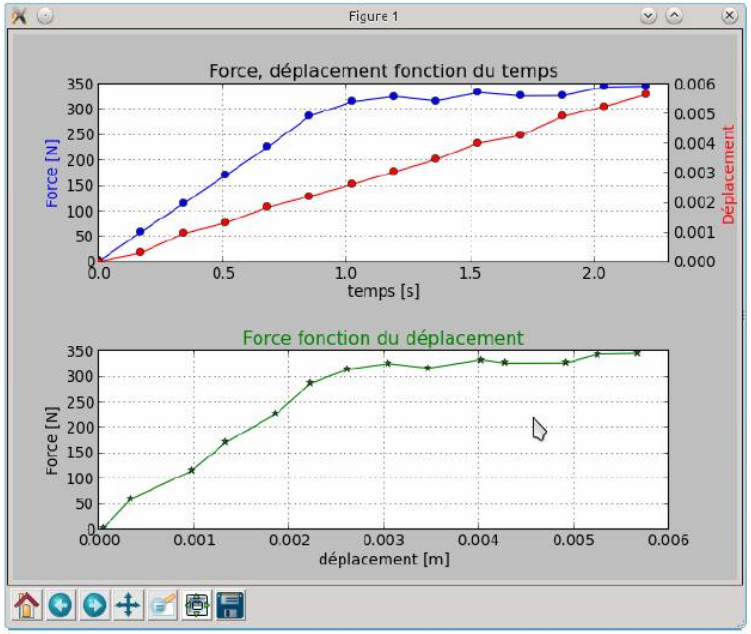
Tracé d'une courbe y=f(x)

[Plot_XY_3.py]

Plusieurs graphes sur une même `figure` :

```
...
plt.figure()
plt.subplots_adjust(hspace=0.5) # ajustement des espaces
plt.subplot(211)          # 2 lignes, 1 col, plot 1 : 211
plt.title("Force, déplacement fonction du temps")
plt.axis([0, 2.3, 0, 350]) ; plt.grid(True)
plt.xlabel("temps [s]")
plt.ylabel("Force [N]", color="b")
plt.plot(t, f,"o-b")

plt.twinx()
plt.axis([0, 2.3, 0, 6.e-3])
plt.ylabel("Déplacement", color="r")
plt.plot(t, d,"o-r")

plt.subplot(212)          # 2 lignes, 1 col, plot 2 : 212
plt.title("Force fonction du déplacement", color="g")
plt.axis([0, 6e-3, 0, 350.0]) ; plt.grid(True)
plt.xlabel("Déplacement [m]")
plt.ylabel("Force [N]")
plt.plot(d, f,"*-g")
```

Tracé d'une courbe y=f(x)



Tracé d'un histogramme

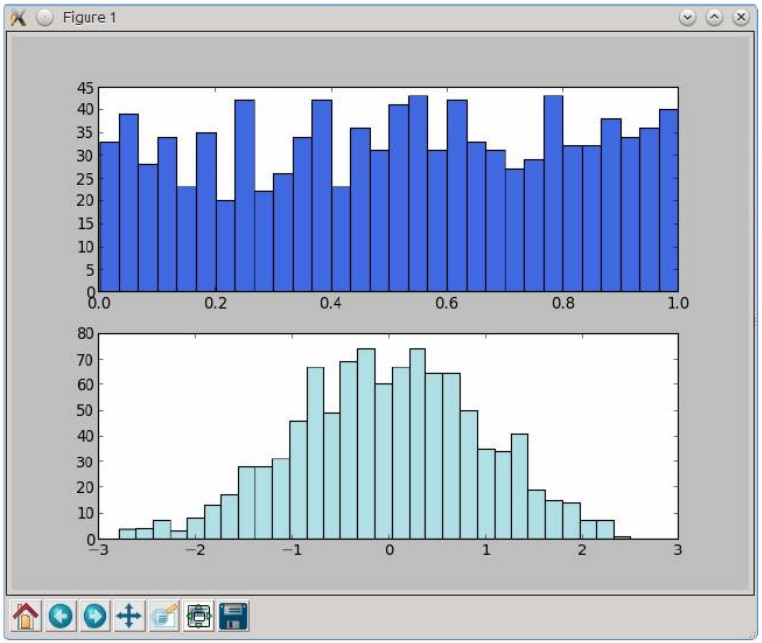
[Plot_Hist_1.py]

```
import numpy as np
import matplotlib.pyplot as plt

y = np.random.rand(1000)  # tirage aleatoire uniforme dans [0,1]
print(type(y))
print(y.shape)

plt.subplot(211)
plt.hist(y, 30, color="RoyalBlue")

y = np.random.randn(1000) # loi normale centree reduite
plt.subplot(212)
plt.hist(y, 30, color="PowderBlue")

plt.show()
```

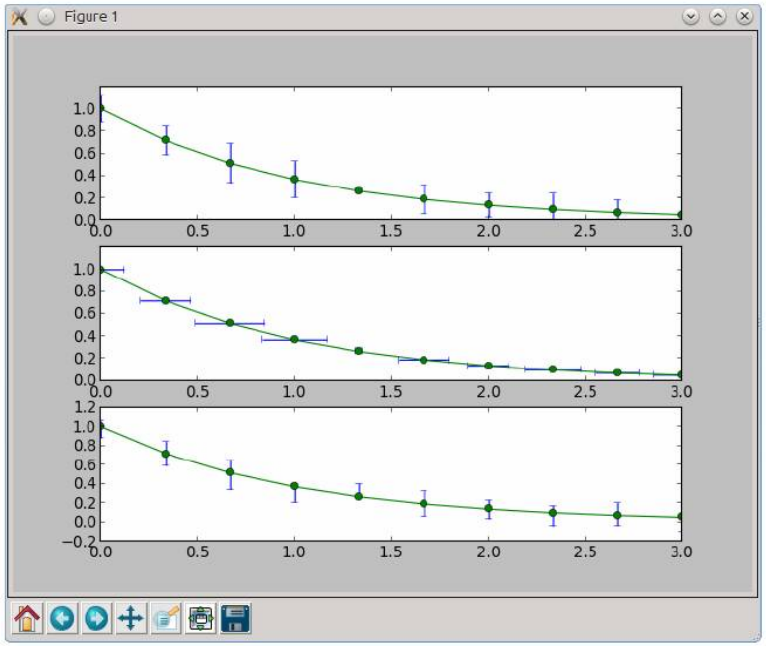
couleurs : `'r','b'` ou `0x10AABE` (*Red, Green, Blue*), ou couleurs HTML

```
<class 'numpy.ndarray'>
(1000,)
```

Tracé d'un histogramme



Tracé d'une courbe avec barres d'erreurs



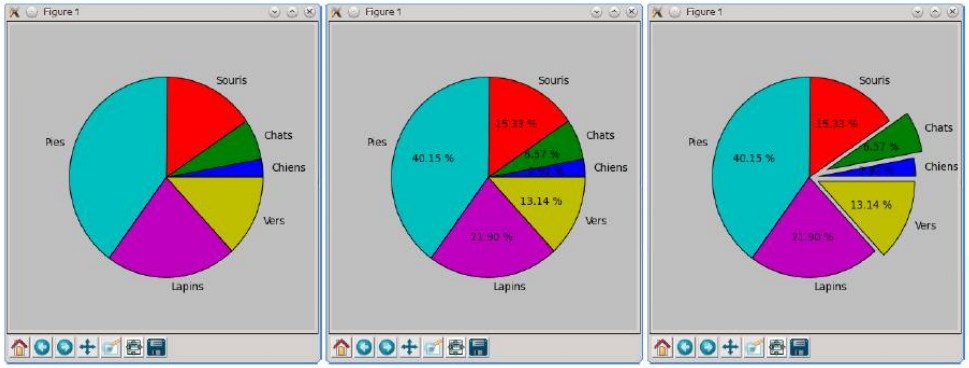
Tracé d'une courbe avec barres d'erreurs [Plot_ErrorBars_1.py]

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 3, 10) # abscisses : 10 pts dans [0, 3]
y = np.exp(-x)            # ordonnées
e1 = 0.2*np.abs(np.random.rand(len(x)))
e2 = 0.2*np.abs(np.random.rand(len(x)))
# barre d"erreur Y symétrique
plt.axis([0, 3, 0, 1.2])
plt.subplot(311)
plt.errorbar(x, y, yerr=e1, fmt="go-", ecolor="b")
# barre d"erreur X symétrique
plt.axis([0, 3, 0, 1.2])
plt.subplot(312)
plt.errorbar(x, y, xerr=e1, fmt="go-", ecolor="b")
# barre d"erreur Y asymétrique
plt.axis([0, 3, 0, 1.2])
plt.subplot(313)
plt.errorbar(x, y, yerr=[e1,e2], fmt="go-", ecolor="b")
plt.show()
```

Tracé d'un 'camembert' [Plot_PieChart_1.py]

```
import numpy as np
import matplotlib.pyplot as plt

x = [4, 9, 21, 55, 30, 18]

plt.figure(figsize=(5,5))
L = ["Chiens", "Chats","Souris", "Pies", "Lapins", "Vers"]
plt.pie(x, labels=L)
plt.show()

# Labels dans les 'portions':
plt.figure(figsize=(5,5))
plt.pie(x, labels=L, autopct="%.2f %%")
plt.show()

# Explode de certaines portions:
e = [0.1, 0.2, 0, 0, 0, 0.1]
plt.figure(figsize=(5,5))
plt.pie(x, labels=L, autopct="%.2f %%", explode=e)
plt.show()
```

Tracé d'un 'camembert'

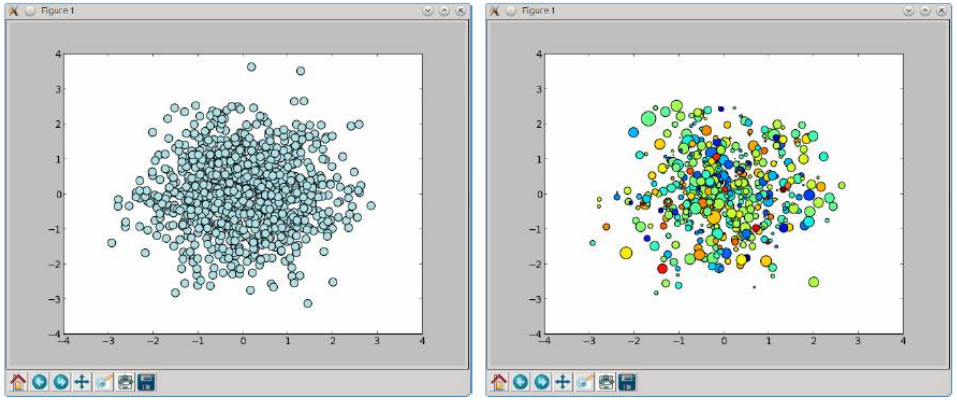


Tracé d'un nuage de points (Scatter)

[Plot_Scatter_1.py]

```
import numpy as np
import matplotlib.pyplot as plt

x = np.random.randn(1000) # 1000 tirages random (loi normale
y = np.random.randn(1000) # centrée réduite)

plt.scatter(x, y, s=100, c="PowderBlue")
plt.show()

sizes =  100*np.random.randn(1000)
colors = np.random.randn(1000)

plt.scatter(x, y, s=sizes, c=colors)
plt.show()
```

Tracé d'un nuage de points (Scatter)

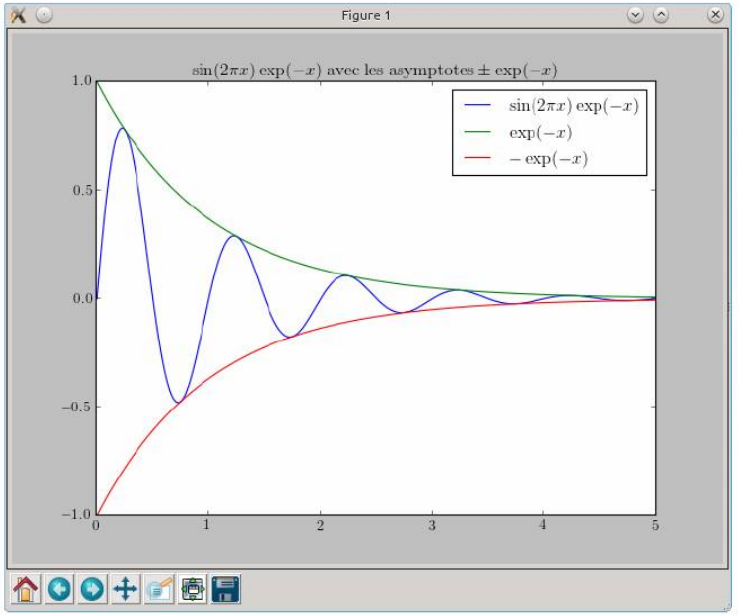


LaTeX dans les Tracés

[Plot_LaTeX_1.py]

```
from numpy import pi, arange, sin, exp

import matplotlib.pyplot as plt
import matplotlib as mpl

# validation de la compilation LaTeX :
mpl.rcParams["text.usetex"] = True

x = arange(0., 5., .01)                  # x values
y = [sin(2*pi*xx)*exp(-xx) for xx in x] # liste par compréhension

# Le texte pour LaTeX est préfixé par "r" (raw) :
plt.plot(x, y, label=r"$\sin(2\pi x)\exp(-x)$")
plt.plot(x,  exp(-x), label=r"$\ \exp(-x)$")
plt.plot(x, -exp(-x), label=r"$-\exp(-x)$")
t1  = r"$\sin(2\pi x)\exp(-x)\ \mathrm{avec\ les\ asymptotes} "
t1 += r"\pm\exp(-x)$"
plt.title(t1)
plt.legend()
plt.show()
```

LaTeX dans les Tracés



Module numpy

La classe ndarray du module numpy

- La fonction `tolist` convertit un objet `ndarray` en objet `list` :

```
>>> m = np.ndarray((3,2))
>>> m.fill(5)
>>> m
array([[ 5.,  5.],
       [ 5.,  5.],
       [ 5.,  5.]])
>>> print(m)
[[ 5.  5.]
 [ 5.  5.]
 [ 5.  5.]]
>>> L = m.tolist() ; L
[[5.0, 5.0], [5.0, 5.0], [5.0, 5.0]]
```

- La fonction `linspace` crée un vecteur de `float` aux coordonnées régulièrement espacées :

```
 >>> np.linspace(0,10,5)
array([ 0. , 2.5, 5. , 7.5, 10. ])
```

- 5 nombres entre 0 et 10 inclus

Attention : comportement différent de `range` ou de `numpy.arange` pour la borne supérieure.

La classe ndarray du module numpy

- Indexation des objets `ndarray` -> comme les objets `list` :

```
>>> m2 = np.array([[ 1.,  2.,  3.],[ 4.,  5., 6.]])
```

```
>>> print(m2)
[[ 1.  2.  3.]
 [ 4.  5.  6.]]
>>> m2[0]
array([ 1.,  2.,  3.])
>>> m2[1]
array([ 4.,  5.,  6.])
>>> m2[0,:]
array([ 1.,  2.,  3.])
>>> m2[:,0]
array([ 1.,  4.])
>>> m2[:,1]
array([ 2.,  5.])
>>> m2[:-1,:-1]
array([[ 1.,  2.]])
```

- `[0]` 1re ligne
- `[1]` 2me ligne
- `[0,:]` 1re ligne, toutes les colonnes
- `[:,0]` toutes les lignes, 1re colonne
- `[:,1]` toutes les lignes, 2me colonne
- `[:-1,:-1]` toutes les lignes sauf la dernière, toutes les colonnes sauf la dernière

Générateurs aléatoires avec numpy.random

- La doc en ligne du module numpy.random est disponible sur http://docs.scipy.org/doc/numpy/reference/routines.random.html
- Exemples de générateurs aléatoires uniformes :
  - np.random.rand tirage uniforme continu dans [0,1[
  - np.random.random tirage uniforme continu dans [0,1[
  - np.random.randint tirage uniforme discret dans [a, b[
  - np.random.random_integers tirage uniforme discret dans [a, b]
  - np.random.randn tirage gaussien dans [0,1[
- Générateurs aléatoires suivant des distributions classiques
  - vaste catalogue (beta, binomial, chisquare, dirichlet, gamma...)

Générateurs aléatoires avec numpy.random

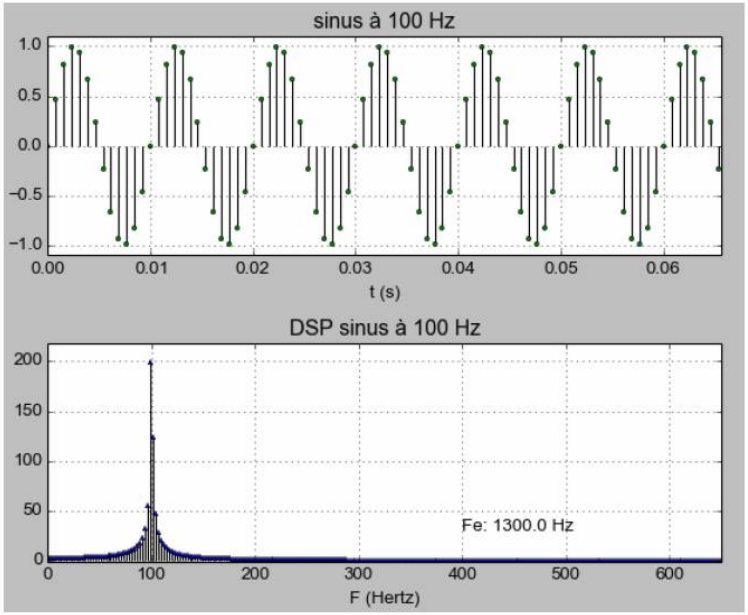
- Générateurs aléatoires (loi uniforme) :

```
>>> import numpy.random as npr
>>> npr.rand(3,2)    # matrice 3X2, tirages uniformes dans [0,1[
array([[ 0.14022471,  0.96360618],
       [ 0.37601032,  0.25528411],
       [ 0.49313049,  0.94909878]])
>>> np.random.randint(5,10, size=10)
array([6, 6, 8, 6, 6, 6, 5, 9, 7, 5])
>>> np.random.randint(5, size=(2, 4)) # tuple pour donner la dimension
array([[4, 0, 2, 4],
       [3, 3, 0, 3]])
>>> np.random.random_integers(1,5,10)# 10 values in [1,5]
array([2, 2, 3, 4, 4, 1, 2, 2, 3, 3])
>>> npr.random()
0.47108547995356098
>>> npr.random((5,))   # tuple pour donner la dimension (shape)
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
>>> 5 * npr.random((3, 2)) - 5        # 3 X 2 array of random reals
array([[-3.99149989, -0.52338984],   # in [-5, 0[
       [-2.99091858, -0.79479508],
       [-1.23204345, -1.75224494]])
```

Calculs de FFT avec numpy.fft [fft_sin100.py]

```
import numpy as np
import matplotlib.pyplot as plt

N, F0 = 512, 100        # N points, 100. Hertz
Fe = 13.*F0             # Fréquence échantillonnage
Te, dF = 1./Fe, Fe/N    # période éch., pas en fréquence
vT = np.arange(N)*Te    # vecteur temps
# échantillonnage du sinus et FFT :
Se  = np.sin(2*np.pi*F0*vT)
TF_Se = np.fft.rfft(Se)
N2 = len(TF_Se); vF = np.arange(N2)*dF
plt.subplots_adjust(hspace=0.4)
plt.subplot(211)
plt.title('sinus à 100 Hz'); plt.xlabel('t (s)'); plt.grid()
plt.plot(vT, Se, 'og', markersize=3)
plt.vlines(vT, [0], Se)
plt.axis([0, N/6*Te, -1.1, 1.1])
plt.subplot(212)
plt.title('DSP sinus à 100 Hz'); plt.xlabel('F (Hertz)'); plt.grid()
plt.plot(vF, abs(TF_Se), '^b', markersize=3)
plt.vlines(vF, [0], abs(TF_Se))
plt.axis([0, Fe/2, -1, 1.1*max(abs(TF_Se))])
plt.text(400, 30, "Fe: {} Hz".format(Fe))
plt.show()
```

Calculs de fft avec numpy.fft



Les classes « objets »

Le concept de classe

Exemple de **définition** de la classe Avare en langage Python :

[Avare.py]

```
class Avare:

    def __init__(self, nom):     # Le constructeur en Python : __init__
        self.__nom = nom         # self.__xx : attribut d'instance privé 'xx'
        self.__fortune = 100000  # why not !!!

    def nom(self):
        return self.__nom

    def encaisser(self, montant):
        self.__fortune += montant

    def depenser(self, montant):
        x = 0.9*montant           # objet (variable) temporaire
        print("je ne peux dépenser que {}".format(x))
        self.__fortune -= x       # avare !

    def compter(self):
        return self.__fortune
```

Relations entre classes

- L'analyse des systèmes (réels ou conceptuels) se fait souvent selon :
  - la **Classification** (approche hiérarchique)
  - la **Composition/Décomposition** (approche structurelle).

"un pommier *est un* arbre *composé* d'un tronc, de branches, de feuilles et de fleurs"
"un élément triangle T6 *est un* élément fini *composé* de 6 noeuds"

- L'approche OO propose plusieurs types de relations entre classes :
  - l'**Héritage**, traduit souvent une classification hiérarchique
  - l'**Agrégation/Composition**, traduit qu'un contenant contient des agrégats/-composants (décomposition structurelle)
  - l'**Association** simple, permet à 2 classes de se connaître
  - l'**Utilisation/Dépendance**, traduit le fait qu'une classe se sert d'une autre.

- Les **relations** entre classes modélisent les relations (les **liens**) entre objets.

Définition d'une classe : mot clef `class`

```
class Point:                # le mot clef 'class' introduit la définition d'une classe

    def __init__(self):     # La méthode spéciale '__init__' joue le rôle du constructeur
        self.x = 0          # les attributs (publics) sont préfixés par self
        self.y = 0

p1 = Point()                # p1 : objet de type Point
print("1:",p1)
p1.x = 1; p1.y = 2;         # attributs publics accessibles en lecture/écriture !
print("2:",p1.x, p1.y)
p1.x = "n'importe quoi"     # danger ....
p1.z = 'ohoh';              # on peut même créer de nouveaux attributs (publics) !!!!
print("3:",p1.x, p1.y, p1.z)
print(dir(p1))              # voir tout ce qu'il y a dans l'objet p1
print(p1.__dict__)          # voir le dictionnaire des attributs de p1
```

```
1: <__main__.Point object at 0x7f6b46e9f748>
2: 1 2
3: n'importe quoi 2 ohoh
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__',
'__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'x', 'y', 'z']
{'z': 'ohoh', 'y': 2, 'x': 1}
```

Accès aux données d'un objet : syntaxe `objet.attribut`

```
class Point:                # le mot clef 'class' introduit la définition d'une classe
    def __init__(self):     # La méthode spéciale '__init__' joue le rôle du constructeur
        self.x = 0          # les attributs (publics) sont préfixés par 'self.'
        self.y = 0

        self.__z = 0        # les attributs "privés" sont préfixés par 'self.__'
                            # self.__z apparaît dans __dict__ comme '_Point__z' (mangling)

p1 = Point()                # p1 : objet de type Point
print(p1.__dict__)

p1.x = 1; p1.y = 2          # attributs publics accessibles en lecture/écriture !

# a = p1.z                  ->  AttributeError: 'Point' object has no attribute 'z'
# a = p1.__z                ->  AttributeError: 'Point' object has no attribute '__z'

p1.__z = 3                  # ERREUR logique ? : crée l'attribut public '__z' !!!
print(p1.__dict__)
```

```
{'_Point__z': 0, 'x': 0, 'y': 0}
{'_Point__z': 0, '__z': 3, 'x': 1, 'y': 2}
```

Accès aux méthodes d'un objet : syntaxe `objet.méthode(...)`

`classe.méthode(objet, args...)` est interprété comme `objet.méthode(args...)`

```
class Point:                # le mot clef 'class' introduit la définition d'une classe

    def __init__(self):     # La méthode spéciale '__init__' joue le rôle du constructeur
        self.x = 0          # les attributs sont préfixés par self
        self.y = 0

    def info(self):         # self est l'objet courant, obligatoire pour une méthode !
        print("x :", self.x, "y :", self.y)

p1 = Point()        # p1 : objet de type Point
p1.info()           # objet.méthode() : exécute 'méthode' avec les données de 'objet'.
print(dir(p1))      # voir tout ce qu'il y a dans l'objet p1
print(p1.__dict__) # les méthodes n'apparaissent pas dans le __dict__ de l'objet
print(Point.__dict__.keys()) # mais dans le __dict__ de la classe
```

```
x : 0 y : 0
['__class__'..., '__weakref__', 'info', 'x', 'y']
{'y': 0, 'x': 0}
dict_keys(['info', '__init__', '__weakref__', '__dict__', '__doc__', '__module__'])
```

Documentation des classes Python

Documentation avec les *docstrings* [class06.py]

```
class A:
    # En présence d'un docstring """...""" ou '''...''' les commmentaires avec des '#'
    # ne sont pas montrés par help(A), cf diapo page suivante...
    """ Exemple de 'Docstring' de la classe A, qui
    peut s'étendre sur plusieurs lignes"""

    def __init__(self):     # méthode spéciale "constructeur"
        """ le constructeur ne prend pas d'argument"""
        print("appel du constructeur de la classe A")
        self.y = 2          # attribut (public) y
        self.__z = 3        # attribut (privé) z

    def obtenir_z(self):
        """ permet de lire la valeur de l'attribut privé z"""
        return self.__z

    def changer_z(self, valeur):
        """ change sous contrôle la valeur de l'attribut privé z"""
        if 0 <= valeur <= 10:
            self.__z = valeur
        else:
            print("uniquement des valeurs entre 0 et 10 SVP !")
```

Redéfinition des opérateurs du langage

Un opérateur Python • écrit `a•b` est interprété comme `a.__•__(b)`

- `__•__` est la **fonction spéciale** associée à l'opérateur •
- `__•__` est redéfinie dans les classes qui doivent supporter l'opérateur •.
- par exemple, pour les opérateurs numériques :
  docs.python.org/reference/datamodel.html#emulating-numeric-types

```
class Point:
    def __init__(self, x=0, y=0):
        self.__x = x      # les attributs privés sont préfixés par 'self.__'
        self.__y = y
    def __eq__(self, p):  # redéfinition de l'opérateur '=='
        return self.__x == p.__x and self.__y == p.__y

p1 = Point(1,1); p2 = Point(2,2); p3 = Point(1,1)
print("p1 == p2 :", p1 == p2); print("p1 == p3 :", p1 == p3)
```

```
p1   ==   p2   : False
p1   ==   p3   : True
```

Propriétés : accès en lecture

- Le décorateur `@property` permet de définir une **propriété** en lecture :
  - accès avec une *syntaxe d'attribut* `a = objet.name`
  - mécanisme interne : la valeur est renvoyée par l'exécution de la méthode associée.

```
from math import sqrt
class Point:
    def __init__(self, x=0, y=0):
        self.__x = x      # les attributs privés sont préfixés par 'self.__'
        self.__y = y
    @property             # Décorateur '@property'
    def distance(self):
        return sqrt(self.__x**2 + self.__y**2)

p1 = Point(1,1)                # p1 : objet de type Point
print("distance:",p1.distance)  # exécution de la méthode associée
# p1.distance = 3               -> AttributeError: can't set attribute
```

```
distance: 1.4142135623730951
```

Propriétés : accès en écriture

- Le décorateur `@name.setter` permet de définir une **propriété** en écriture :
  - accès avec une syntaxe d'attribut `objet.name = a`
  - mécanisme interne : exécution de la méthode associée

```
from math import sqrt
class Point:
    def __init__(self, x=0, y=0):
        self.__x, self.__y = x, y  # attributs privés préfixés par 'self.__'
    @property
    def xvalue(self): return self.__x
    @xvalue.setter
    def xvalue(self, value): self.__x = value

p1 = Point(1,1)             # p1 : objet de type Point
print("xvalue:",p1.xvalue) # exécution de la méthode associée @property
p1.xvalue = 3               # exécution de la méthode associée par @xvalue.setter
print("xvalue:",p1.xvalue)
```

```
xvalue: 1
xvalue: 3
```

Héritage : accès aux données de la classe de base

```
class A():
    def __init__(self):     # Constructeur de la classe A :
        self.a = ['A']      #  définition de a, attribut public de A
        self.__x = -1       #  définition de x, attribut privé de A.

class B(A):
    def __init__(self):     # Constructeur de la classe B :
        super().__init__()  #  exécution du constructeur de la classe de base
        self.b = 0          #  définition de b : attribut public de B
        self.a.append('B')  #  utilisation de a : attribut public hérité de A
        # c = self.__x      -> ERREUR : x est privé dans la classe A !

class C(A):
    def __init__(self):   # Constructeur de la classe C :
        A.__init__(self)  #  exéc. constructeur de la classe A (autre syntaxe)
        self.a=2          #  définition de a : attribut public de C, masque A.a !

b = B(); print(b.__dict__)
c = C(); print(c.__dict__)
```

```
{'a': ['A', 'B'], '_A__x': -1, 'b': 0}
{'a': 2, '_A__x': -1}
```

Héritage : surcharge/accès aux méthodes de la classe de base

```
class A():
    def go(self): print("A is running go()")

class B(A):
    def go(self): print("B is running go()")

class C(B):
    def go(self):
        A.go(self); B.go(self)

class D(B): pass

if __name__ == "__main__":
    a = A(); print(" a.go() gives: ",end=""); a.go()
    b = B(); print(" b.go() gives: ",end=""); b.go()
    c = C(); print(" c.go() gives: ",end=""); c.go()
    d = D(); print(" d.go() gives: ",end=""); d.go()
```

```
 a.go() gives: A is running go()
 b.go() gives: B is running go()
 c.go() gives: A is running go()
B is running go()
 d.go() gives: B is running go()
```

Accès aux données d'un objet (avancé)

Comment l'interpréteur Python "interprète" l'accès aux données...

- `obj.name = value` est traduit par `obj.__setattr__("name", value)`
- `del obj.name` est traduit par `obj.__delattr__("name")`
- Comportement par défaut de `__setattr__` et `__getattr__` :
  - utiliser d'abord le dictionnaire `__dict__` des attributs de `obj` ;
  - si `name` est une **propriété** (*property*) : les opérations `set` et `delete` sont alors réalisées par celles des propriétés.
- Évaluer `obj.name` conduit à rechercher `obj.__getattribute__("name")`
- Comportement par défaut de `__getattribute__` :
  - rechercher `'name'` dans les **propriétés**, le `__dict__` local, puis les classes de base.

Attributs de classe (statiques)

```
class Point:
    liste  = []      # définition de l'attribut de classe (statique) public 'liste'
    __nbPt = 0       # définition de l'attribut de classe (statique) privé 'nbpt'

    def __init__(self, x=0, y=0):
        self.__x = x       # définition d'un attribut d'instance -> préfixé par 'self.'
        self.__y = y
        Point.__nbPt += 1        # utilisation d'un attribut statiques : préfixer son
        Point.liste.append(self) # nom par le nom de la classe.

    def __str__(self):           # redéfinition (surcharge) de la fonction str()
        return "x: {0:f}, y: {1:f}".format(self.__x, self.__y)

Point(1,1); Point(2,2)  # création d'objets Point sans nom
p1=Point(3,3)
print(len(Point.liste)) # l'attribut statique 'liste' est public
print(Point.liste[0]); print(Point.liste[1])
# print(Point.__nbPt)   -> ERREUR : attribut statique 'nbPt' est privé !
print(len(p1.liste))    # l'attribut statique punlic 'liste' peut être accédé
                        # en préfixant avec un objet de la classe Point.
```

```
3
x: 1.000000, y: 1.000000
x: 2.000000, y: 2.000000
3
```

Méthodes statiques

```
class Point:
    __liste = []    # définition de l'attribut de classe (statique) public 'liste'

    def __init__(self, x=0, y=0):
        self.__x = x  # attribut privé
        self.__y = y  # attribut privé
        Point.__liste.append(self) # utilisation d'un attribut statiques : préfixer
                                   # son nom par le nom de la classe.

    def __str__(self):       # redéfinition (surcharge) de la fonction str()
        return "x: {0:f}, y: {1:f}".format(self.__x, self.__y)

    @staticmethod            # le décorateur @staticmethod permet de définir
    def nbPoint():           # les méthodes statiques => Pas d'argument self !
        return len(Point.__liste)

Point(1,1)                   # création de 2 objets Point sans nom
Point(2,2)
p1=Point(3,3)                # création d'un objet Point nommé p1
# print(len(Point.__liste))  -> ERREUR : attribut statique 'nbPt' est privé !
print(Point.nbPoint())       # appel de la méthode statique
```

```
3
```