
Une introduction au langage python

Lycée Thiers

2014 - 15

Jean-Philippe Préaux

Table des matières

1	Premier contact	5
1.1	Introduction	5
1.1.1	Le langage python	5
1.2	Utilisation en mode console	5
1.2.1	Lancement en mode console	5
1.2.2	Calculatrice	8
1.2.3	Le module <code>math</code>	14
1.2.4	Définition d'une fonction	15
1.2.5	Variables	16
2	Premiers programmes	23
2.1	Diverses utilisations de <code>python</code>	23
2.1.1	Ecriture d'un programme <code>python</code>	23
2.1.2	Environnement de programmation	23
2.2	Programmation en <code>python</code>	26
2.2.1	Saisie de données par l'utilisateur : <code>input()</code>	26
2.2.2	Un nouveau type de variables : <code>booléen</code>	27
2.2.3	Structures de contrôle	29
2.2.4	Structure de boucle <code>while</code>	29
2.2.5	Structure de test	32
3	Listes et boucle for	37
3.1	Les listes en <code>python</code>	37
3.1.1	Les listes	37
3.1.2	Création d'un itérateur avec <code>range()</code>	40
3.2	la boucle <code>for</code>	41
3.2.1	Boucle <code>for</code>	41
3.2.2	Exemples	42
4	Listes et autres conteneurs	47
4.1	Approfondissements sur les listes	47
4.1.1	Les méthodes de la classe <code>list</code>	47
4.1.2	Saucissonnage ou slicing	47
4.1.3	Copie de liste	49

4.1.4	Liste définie par compréhension	50
4.2	Structures de données séquentielles	51
4.2.1	Opérations communes	51
4.2.2	Les chaînes de caractère	52
4.2.3	les <code>tuple</code> ou <code>sequence</code>	53
4.3	Les dictionnaires	54
4.3.1	Les dictionnaires	54
4.3.2	Opérations sur les dictionnaires	54
4.3.3	Exemple	55
5	Les modules	57
5.1	Utilisation d'un module	57
5.1.1	Importer un module	57
5.1.2	Exemple : le module <code>random</code>	58
5.2	<code>numpy</code> et <code>matplotlib</code>	59
5.2.1	Modules scientifiques	59
5.2.2	Le module <code>numpy</code>	59
5.2.3	Le module <code>matplotlib</code>	63
5.3	<code>scipy</code>	68
5.3.1	Intégration d'équations différentielle sous <code>scipy</code>	70
5.3.2	Equation différentielles d'ordres supérieurs	71
A	python 2 versus python 3	75
A.1	<code>python</code> versions 2 et 3	75
A.2	Principaux changements de la version 3	76
A.2.1	Changements de la fonction <code>print()</code>	76
A.2.2	Changements de la fonction <code>input()</code>	77
A.2.3	Changement de l'opération de division <code>/</code>	77
A.2.4	Changements de la fonction <code>range()</code>	78

Chapitre 1

Premier contact

1.1 Introduction

1.1.1 Le langage python

Le langage python, qu'est-ce que c'est ?

Python est un langage de programmation impérative, structurée, orientée objet, de haut niveau.

Il présente les avantages suivants :

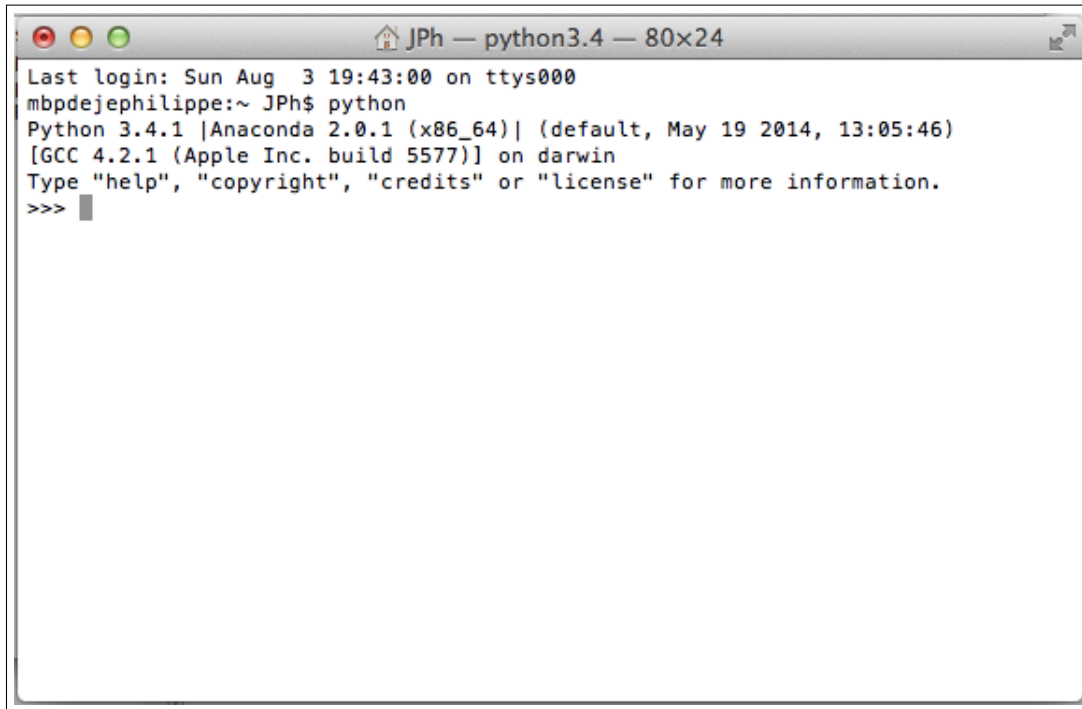
- Sa syntaxe est très simple et concise : "on code ce que l'on pense". Donc facile à apprendre. Proche du 'langage algorithmique'.
- Moderne. Très largement répandu dans l'industrie, l'enseignement et la recherche, notamment pour ses applications scientifiques. Une large communauté participe à son développement.
- Puissant, muni de nombreuses bibliothèques de fonctions. Dont de très bonnes bibliothèques scientifiques.
- Pratique pour travailler sur des objets mathématiques. Assez proche du langage mathématique.
- Gratuit, disponible sur la plupart des plateformes (Windows, Mac, Linux, ...).

1.2 Utilisation en mode console

1.2.1 Lancement en mode console

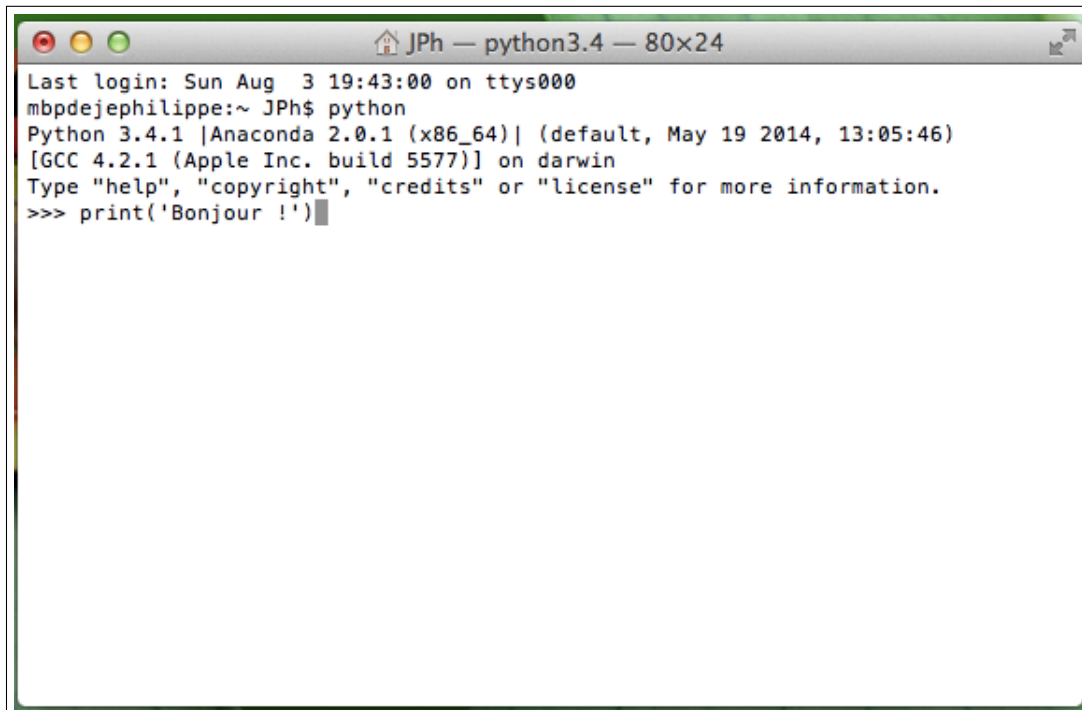
Python utilisé en mode console

Python est un langage interprété qui peut être utilisé en mode console. Lancer une fenêtre de console (terminal, console ou fenêtre de commande selon le Système d'exploitation : `cmd.exe` sous Windows) et simplement taper à l'invite 'python' suivi de la touche entrée.

A terminal window titled 'JPh — python3.4 — 80x24'. The window shows the output of the 'python' command. The text displayed is: 'Last login: Sun Aug 3 19:43:00 on ttys000', 'mbpdejephilippe:~ JPh\$ python', 'Python 3.4.1 [Anaconda 2.0.1 (x86_64)] (default, May 19 2014, 13:05:46)', '[GCC 4.2.1 (Apple Inc. build 5577)] on darwin', 'Type "help", "copyright", "credits" or "license" for more information.', and '>>>' followed by a cursor. The window has standard macOS window controls (red, yellow, green buttons) in the top-left corner.

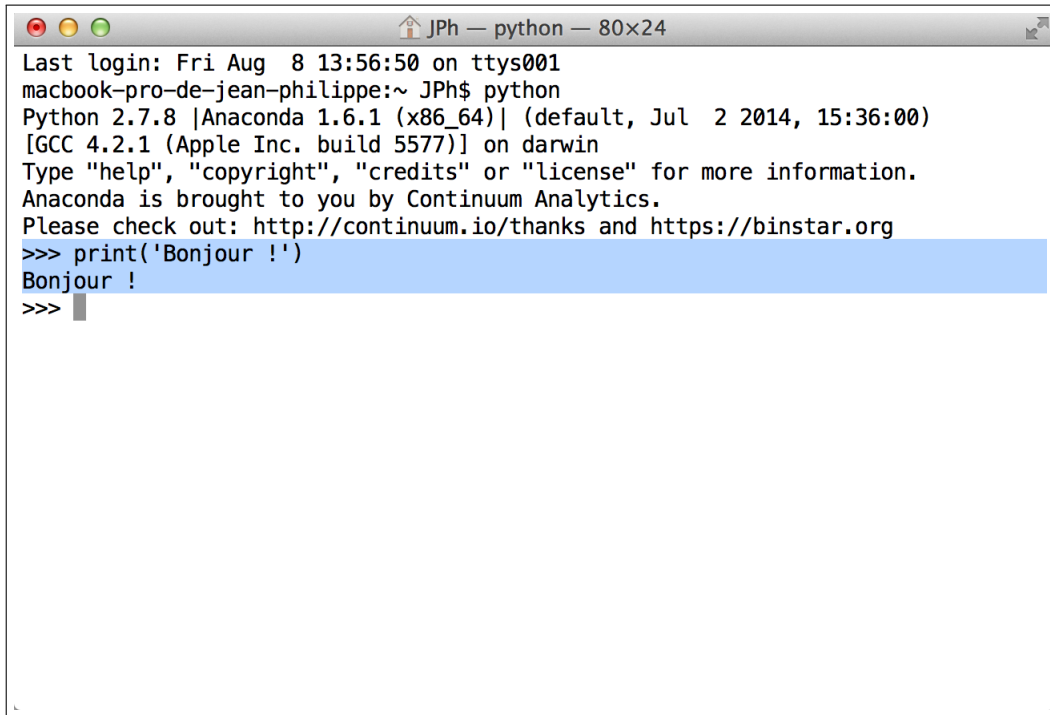
```
Last login: Sun Aug 3 19:43:00 on ttys000
mbpdejephilippe:~ JPh$ python
Python 3.4.1 [Anaconda 2.0.1 (x86_64)] (default, May 19 2014, 13:05:46)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> █
```

Une console au lancement de python. Un prompt, symbolisé ici par : >>>, apparaît.

A terminal window titled 'JPh — python3.4 — 80x24'. The window shows the same output as the previous image, but with an additional line of code entered at the prompt: '>>> print('Bonjour !')' followed by a cursor. The window has standard macOS window controls (red, yellow, green buttons) in the top-left corner.

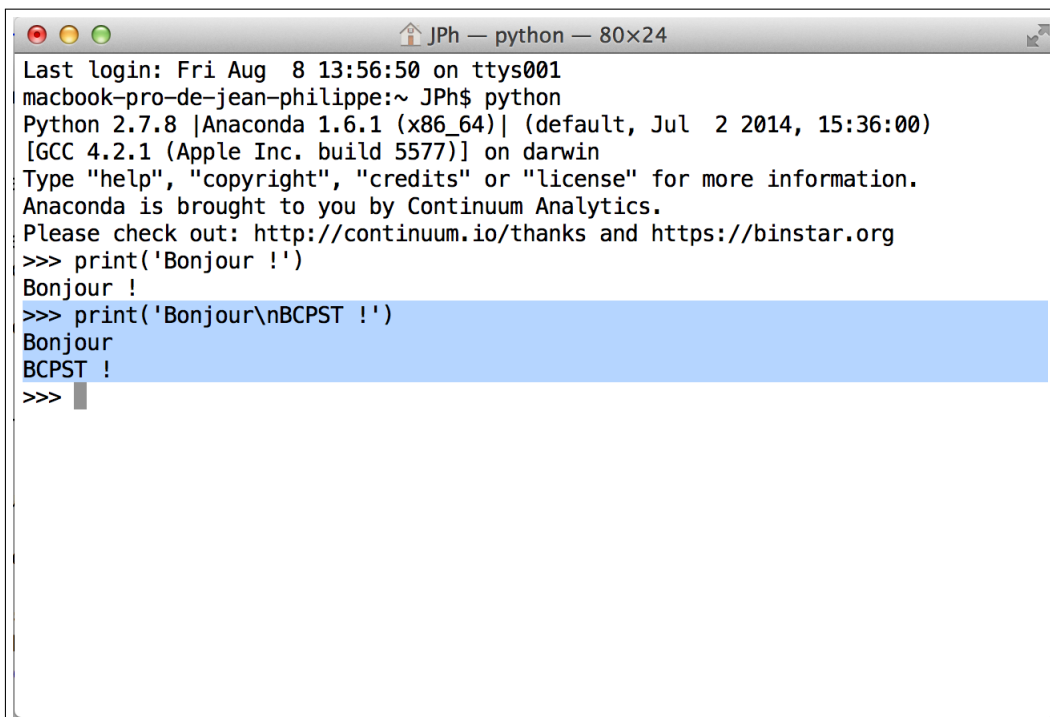
```
Last login: Sun Aug 3 19:43:00 on ttys000
mbpdejephilippe:~ JPh$ python
Python 3.4.1 [Anaconda 2.0.1 (x86_64)] (default, May 19 2014, 13:05:46)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Bonjour !') █
```

Une instruction saisie au prompt est lancée en appuyant sur la touche ENTREE.

A terminal window titled 'JPh — python — 80x24' with standard macOS window controls. The terminal shows the output of running 'python' on a Mac. It displays the last login time, the Python version (2.7.8) and Anaconda version (1.6.1), and the GCC version (4.2.1). It then prompts for help information. The user enters 'print('Bonjour !')' and the terminal outputs 'Bonjour !'.

```
Last login: Fri Aug 8 13:56:50 on ttys001
macbook-pro-de-jean-philippe:~ JPh$ python
Python 2.7.8 |Anaconda 1.6.1 (x86_64)| (default, Jul 2 2014, 15:36:00)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://binstar.org
>>> print('Bonjour !')
Bonjour !
>>>
```

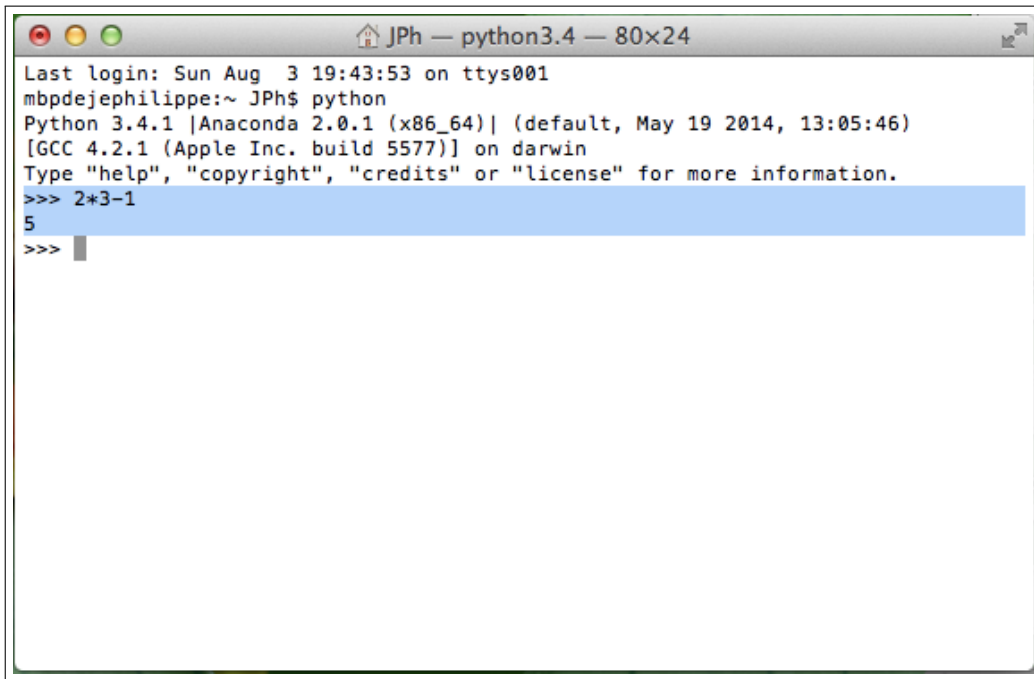
La commande (ou fonction) 'print(.)' écrit à l'écran une chaîne de caractère, c'est à dire une suite finie de caractères entre apostrophes '...' ou double-quotes "...".

A terminal window titled 'JPh — python — 80x24' with standard macOS window controls. The terminal shows the output of running 'python' on a Mac. It displays the last login time, the Python version (2.7.8) and Anaconda version (1.6.1), and the GCC version (4.2.1). It then prompts for help information. The user enters 'print('Bonjour !')' and the terminal outputs 'Bonjour !'. The user then enters 'print('Bonjour\nBCPST !')' and the terminal outputs 'Bonjour' followed by 'BCPST !' on a new line.

```
Last login: Fri Aug 8 13:56:50 on ttys001
macbook-pro-de-jean-philippe:~ JPh$ python
Python 2.7.8 |Anaconda 1.6.1 (x86_64)| (default, Jul 2 2014, 15:36:00)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://binstar.org
>>> print('Bonjour !')
Bonjour !
>>> print('Bonjour\nBCPST !')
Bonjour
BCPST !
>>>
```

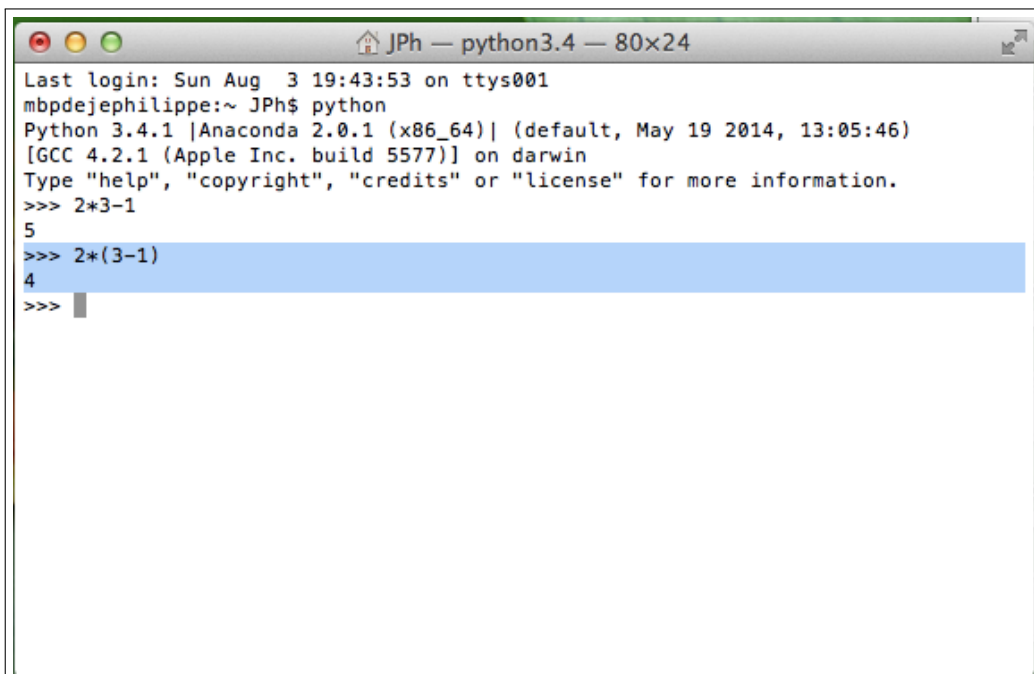
Le caractère spécial `\n` permet un passage à la ligne.

1.2.2 Calculatrice



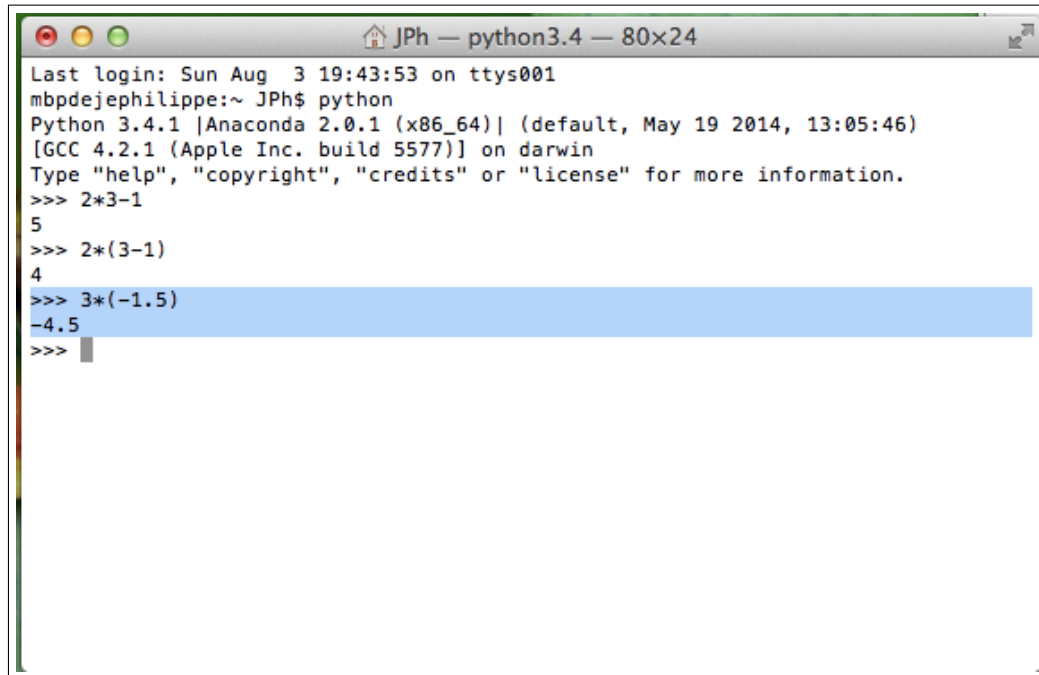
```
JPh — python3.4 — 80x24
Last login: Sun Aug  3 19:43:53 on ttys001
mbpdejephilippe:~ JPh$ python
Python 3.4.1 [Anaconda 2.0.1 (x86_64)] (default, May 19 2014, 13:05:46)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 2*3-1
5
>>>
```

Python peut se comporter comme une calculatrice.



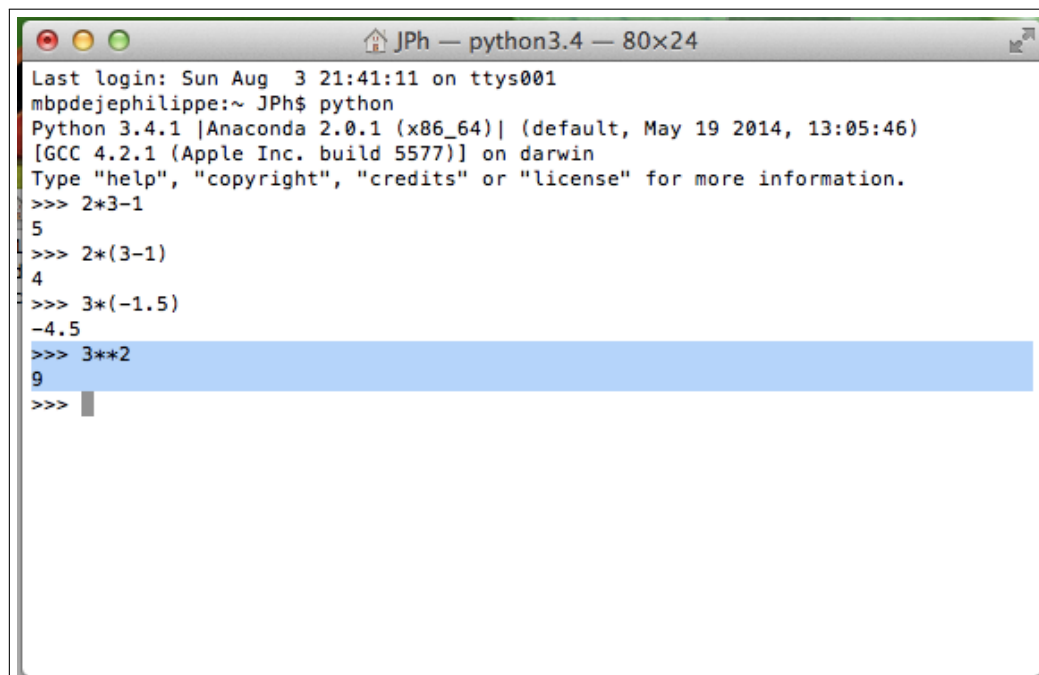
```
JPh — python3.4 — 80x24
Last login: Sun Aug  3 19:43:53 on ttys001
mbpdejephilippe:~ JPh$ python
Python 3.4.1 [Anaconda 2.0.1 (x86_64)] (default, May 19 2014, 13:05:46)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 2*3-1
5
>>> 2*(3-1)
4
>>>
```

Il respecte l'ordre usuel des opérations.



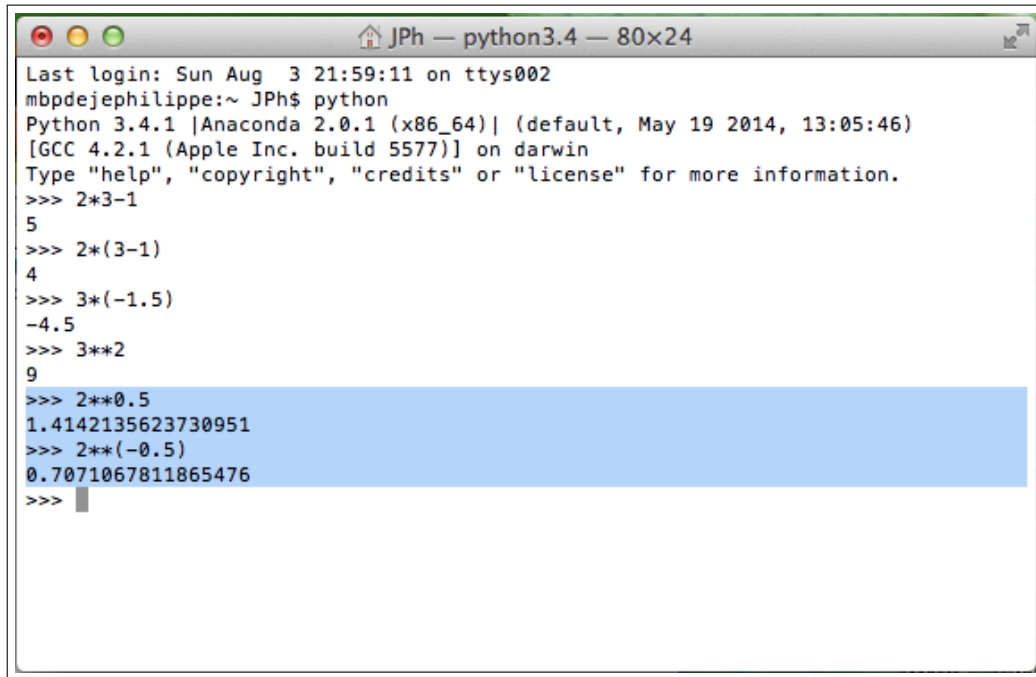
```
JPh — python3.4 — 80x24
Last login: Sun Aug  3 19:43:53 on ttys001
mbpdejephilippe:~ JPh$ python
Python 3.4.1 |Anaconda 2.0.1 (x86_64)| (default, May 19 2014, 13:05:46)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 2*3-1
5
>>> 2*(3-1)
4
>>> 3*(-1.5)
-4.5
>>>
```

Et comprend les nombres à virgule flottante 'de type float'.



```
JPh — python3.4 — 80x24
Last login: Sun Aug  3 21:41:11 on ttys001
mbpdejephilippe:~ JPh$ python
Python 3.4.1 |Anaconda 2.0.1 (x86_64)| (default, May 19 2014, 13:05:46)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 2*3-1
5
>>> 2*(3-1)
4
>>> 3*(-1.5)
-4.5
>>> 3**2
9
>>>
```

La mise en puissance s'obtient grâce à l'opérateur `**`.

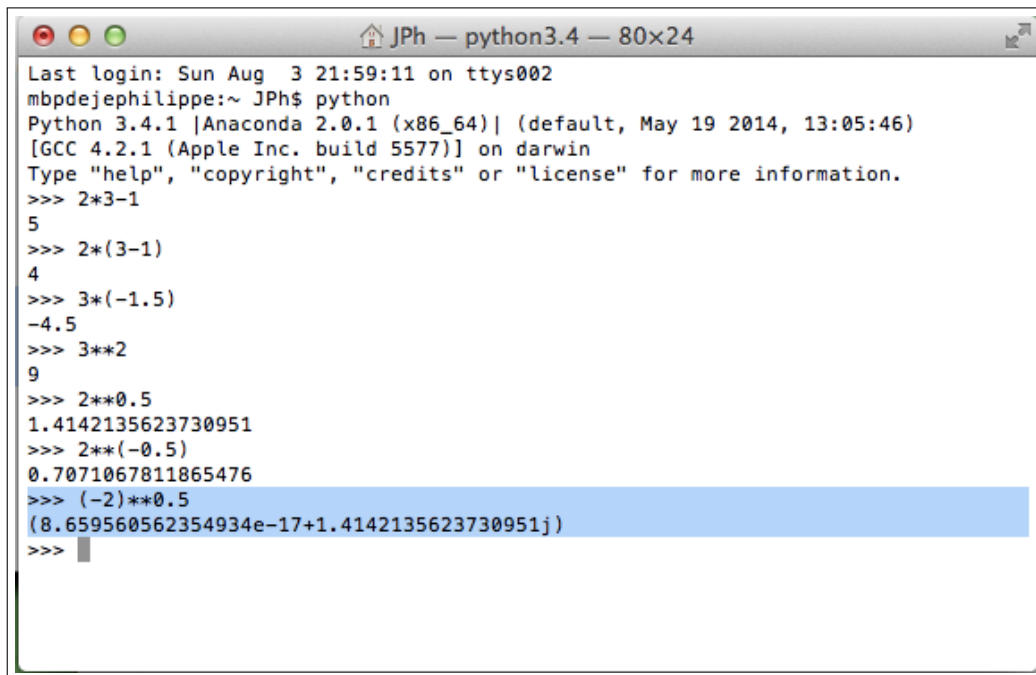


```

Last login: Sun Aug  3 21:59:11 on ttys002
mbpdejephilippe:~ JPh$ python
Python 3.4.1 [Anaconda 2.0.1 (x86_64)] (default, May 19 2014, 13:05:46)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 2*3-1
5
>>> 2*(3-1)
4
>>> 3*(-1.5)
-4.5
>>> 3**2
9
>>> 2**0.5
1.4142135623730951
>>> 2**(-0.5)
0.7071067811865476
>>> 

```

L'exposant peut être 'réel'. Ainsi l'opérateur `**0.5` extrait la racine carrée.

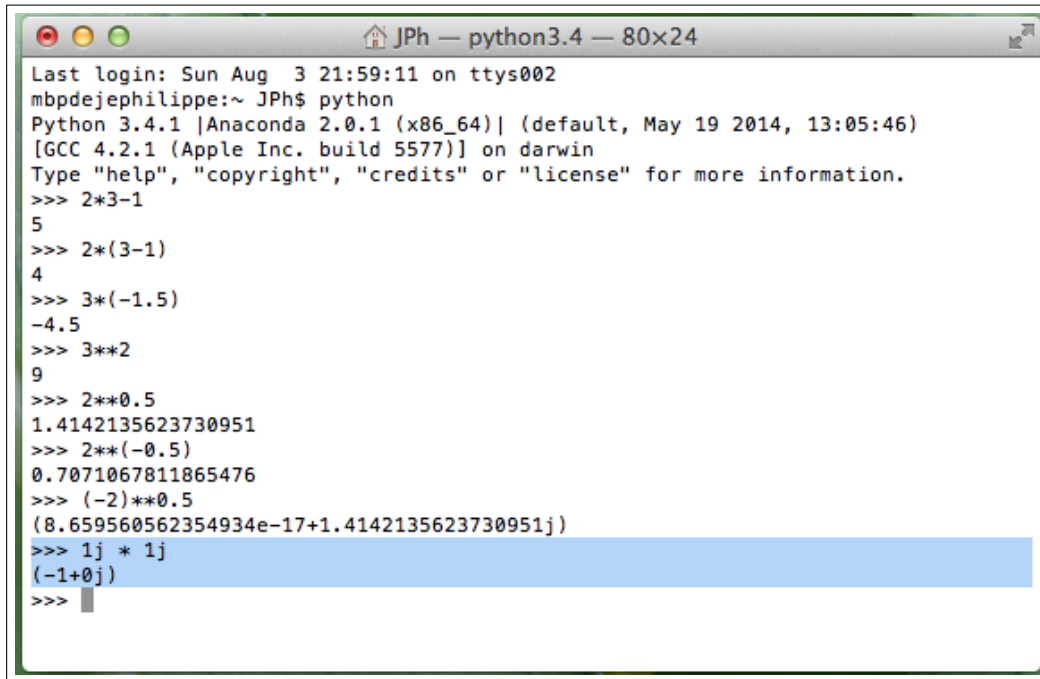


```

Last login: Sun Aug  3 21:59:11 on ttys002
mbpdejephilippe:~ JPh$ python
Python 3.4.1 [Anaconda 2.0.1 (x86_64)] (default, May 19 2014, 13:05:46)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 2*3-1
5
>>> 2*(3-1)
4
>>> 3*(-1.5)
-4.5
>>> 3**2
9
>>> 2**0.5
1.4142135623730951
>>> 2**(-0.5)
0.7071067811865476
>>> (-2)**0.5
(8.659560562354934e-17+1.4142135623730951j)
>>> 

```

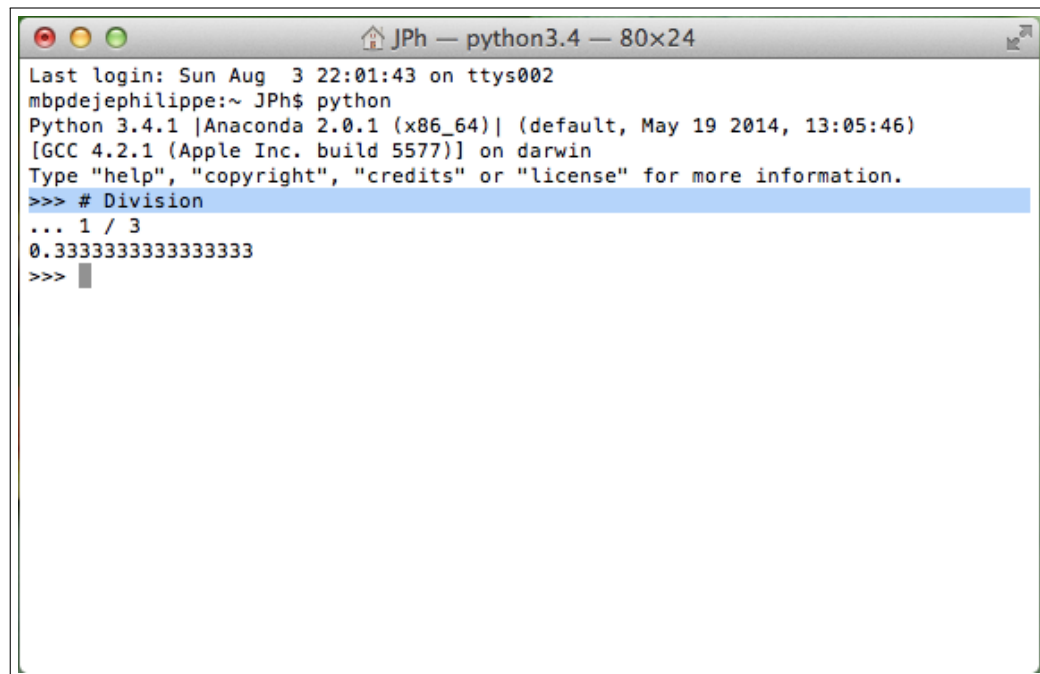
Lorsque y n'est pas entier, le réel x^y n'est défini que pour $x > 0$: `python` (version 3) retourne un nombre complexe, valeur approchée ici de $i\sqrt{2}$.



```
Last login: Sun Aug 3 21:59:11 on ttys002
mbpdejephilippe:~ JPh$ python
Python 3.4.1 |Anaconda 2.0.1 (x86_64)| (default, May 19 2014, 13:05:46)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 2*3-1
5
>>> 2*(3-1)
4
>>> 3*(-1.5)
-4.5
>>> 3**2
9
>>> 2**0.5
1.4142135623730951
>>> 2**(-0.5)
0.7071067811865476
>>> (-2)**0.5
(8.659560562354934e-17+1.4142135623730951j)
>>> 1j * 1j
(-1+0j)
>>>
```

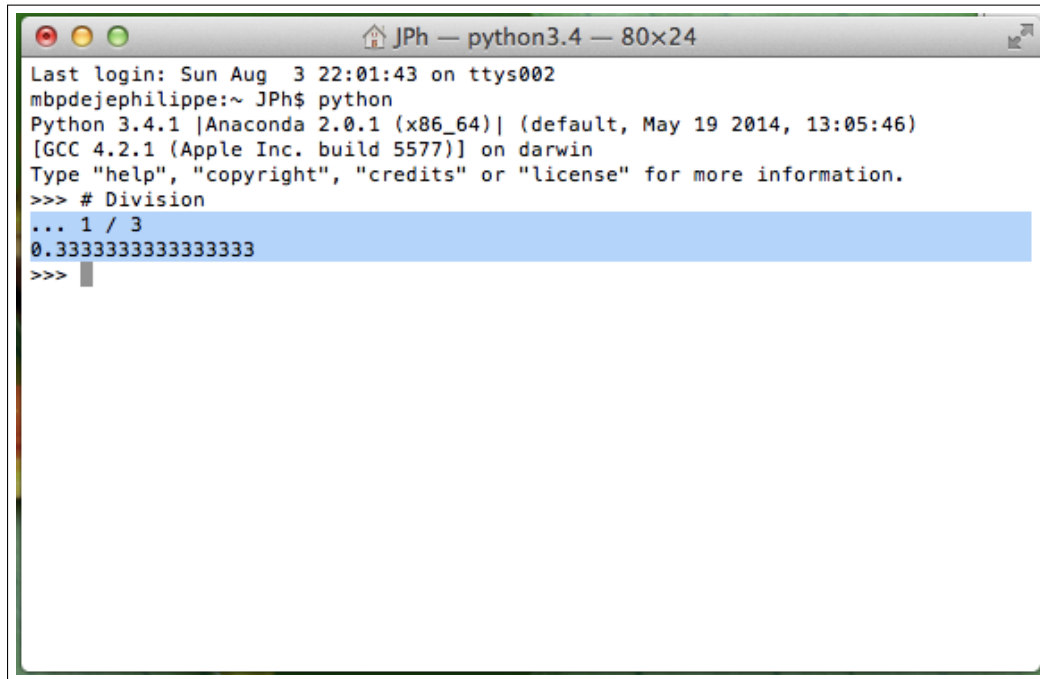
$x + i.y$ s'écrit $x+yj$ (avec x,y des flottants).

Remarque : opérandes et opérations peuvent être séparés d'aucun, un, ou plusieurs espaces.



```
Last login: Sun Aug 3 22:01:43 on ttys002
mbpdejephilippe:~ JPh$ python
Python 3.4.1 |Anaconda 2.0.1 (x86_64)| (default, May 19 2014, 13:05:46)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> # Division
... 1 / 3
0.3333333333333333
>>>
```

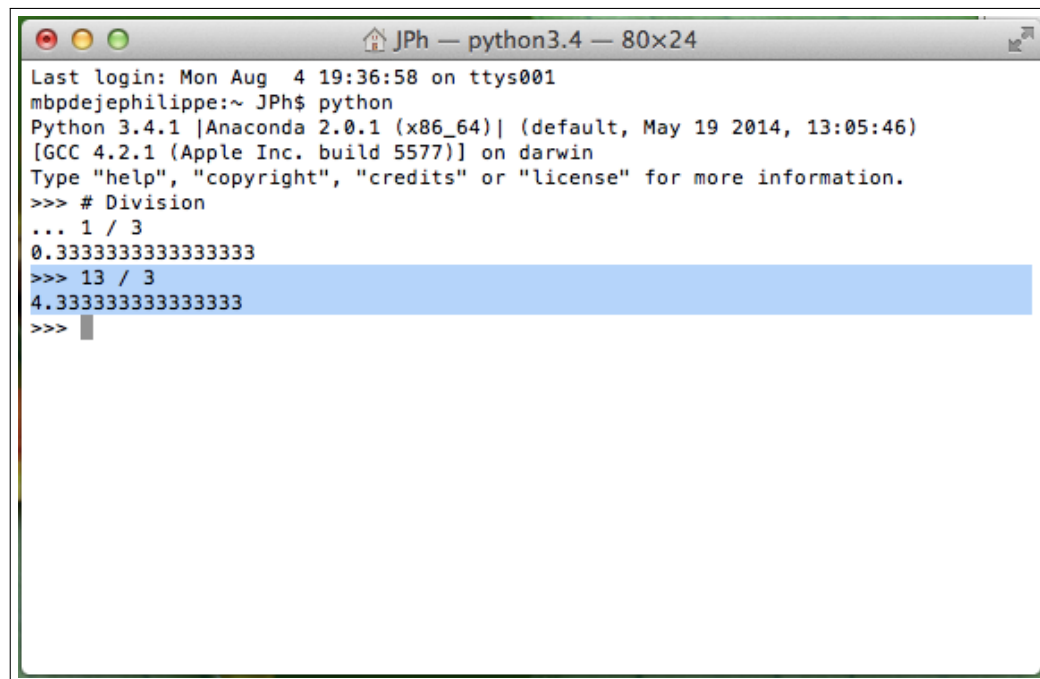
Les commentaires sont placés après un symbole `#`. Tout ce qui est placé après un symbole `#` sur une ligne est ignoré par l'interpréteur. Il est essentiel de les utiliser lors de l'écriture d'un programme.



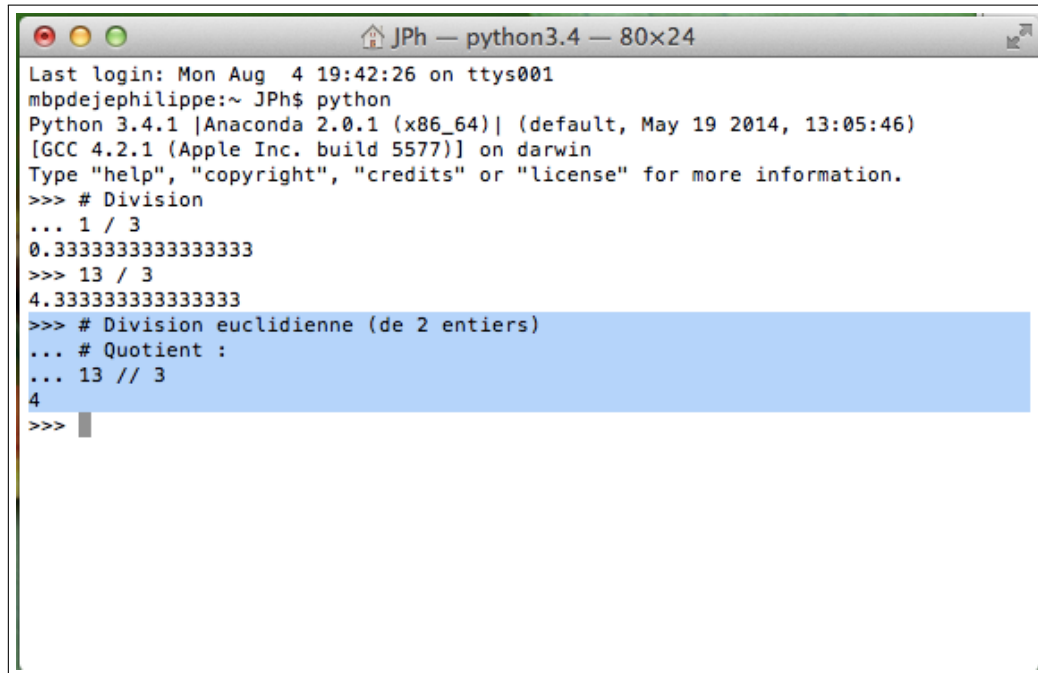
```
JPh — python3.4 — 80x24
Last login: Sun Aug 3 22:01:43 on ttys002
mbpdejephilippe:~ JPh$ python
Python 3.4.1 [Anaconda 2.0.1 (x86_64)] (default, May 19 2014, 13:05:46)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> # Division
... 1 / 3
0.3333333333333333
>>>
```

/ est l'opérateur de division.

Attention son comportement est différent dans la version 2 de `python` lorsque ses opérandes sont entiers (il se comporte comme // en `python 3` décrits ci-dessous ; voir l'annexe).

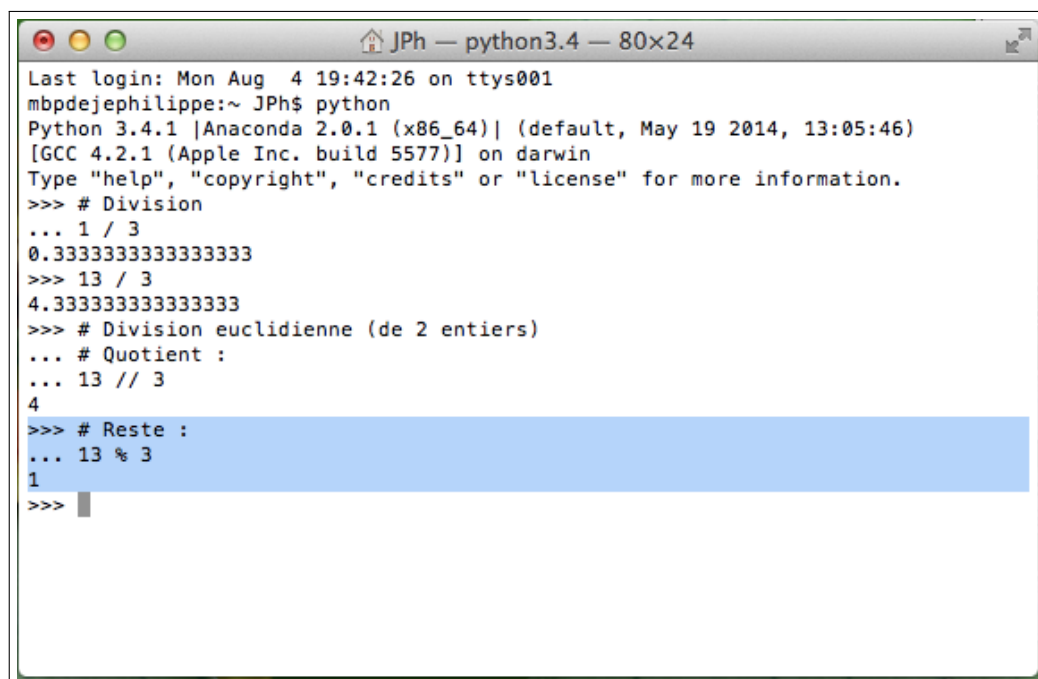


```
JPh — python3.4 — 80x24
Last login: Mon Aug 4 19:36:58 on ttys001
mbpdejephilippe:~ JPh$ python
Python 3.4.1 [Anaconda 2.0.1 (x86_64)] (default, May 19 2014, 13:05:46)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> # Division
... 1 / 3
0.3333333333333333
>>> 13 / 3
4.333333333333333
>>>
```



```
JPh — python3.4 — 80x24
Last login: Mon Aug  4 19:42:26 on ttys001
mbpdejephilippe:~ JPh$ python
Python 3.4.1 [Anaconda 2.0.1 (x86_64)] (default, May 19 2014, 13:05:46)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> # Division
... 1 / 3
0.3333333333333333
>>> 13 / 3
4.333333333333333
>>> # Division euclidienne (de 2 entiers)
... # Quotient :
... 13 // 3
4
>>>
```

// est l'opérateur de quotient de 2 entiers n et $m \neq 0$ dans la division euclidienne de n par m ; il retourne le quotient entier $n//m = \lfloor \frac{n}{m} \rfloor$.



```
JPh — python3.4 — 80x24
Last login: Mon Aug  4 19:42:26 on ttys001
mbpdejephilippe:~ JPh$ python
Python 3.4.1 [Anaconda 2.0.1 (x86_64)] (default, May 19 2014, 13:05:46)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> # Division
... 1 / 3
0.3333333333333333
>>> 13 / 3
4.333333333333333
>>> # Division euclidienne (de 2 entiers)
... # Quotient :
... 13 // 3
4
>>> # Reste :
... 13 % 3
1
>>>
```

L'opérateur % retourne le reste de la division euclidienne : $n = (n//m) \times m + (n\%m)$.

```

Last login: Mon Aug  4 19:42:26 on ttys001
mbpdejephilippe:~ JPh$ python
Python 3.4.1 |Anaconda 2.0.1 (x86_64)| (default, May 19 2014, 13:05:46)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> # Division
... 1 / 3
0.3333333333333333
>>> 13 / 3
4.333333333333333
>>> # Division euclidienne (de 2 entiers)
... # Quotient :
... 13 // 3
4
>>> # Reste :
... 13 % 3
1
>>> # Vérification :
... 3 * 4 + 1
13
>>>

```

L'opérateur % retourne le reste de la division euclidienne : $n = (n//m) \times m + (n\%m)$.

1.2.3 Le module math

le module math

Par défaut python ne connaît, en dehors des opérateurs arithmétiques, aucune fonction ou constante mathématiques : sans bibliothèque l'appel de `cos(1)` ou `pi` produit une erreur.

```

>>> # Sans bibliothèque python est ignorant en math :
... cos(1)
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: name 'cos' is not defined
>>> pi
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pi' is not defined

```

Il suffit de faire appel à la bibliothèque 'math' de fonctions mathématiques prédéfinies. C'est un *module* :

```

>>> from math import *          # importation des fonctions de la bibliothèque
>>> pi
3.141592653589793
>>> cos(pi)
-1.0
>>> acos(-1)
3.141592653589793

```

Le module math

```
>>> sqrt(2)                # racine carrée
1.4142135623730951
>>> e
2.718281828459045
>>> log(e)                 # logarithme neperien
1.0
>>> exp(1)                 # exponentielle
2.718281828459045
>>> log(256,2)             # logarithme en base 2
8.0
>>> log(1000,10)           # logarithme en base 10
2.9999999999999996
>>> # lui préférer :
... log10(1000)            # logarithme base 10 plus précis
3.0
_
>>> fabs(-3)               # valeur absolue
3.0
>>> floor(pi)              # partie entière
3.0
>>> floor(-pi)
-4.0
```

1.2.4 Définition d'une fonction

Définition de fonction

L'utilisateur peut définir ses propres *fonctions* :

```
>>> # Definition d'une FONCTION
... def maFonction(x):
...     return x**2-2*x+1
...
>>> maFonction(1)
0
>>> maFonction(0)
1
```

Ici l'appel de `maFonction(x)` retourne $x^2 - 2x + 1$ grâce à l'instruction `return`.

Une fonction peut aussi ne retourner aucun résultat, c'est une *procédure* :

```
>>> # Définition d'un procédure (pour python c'est aussi une fonction)
... def maProcédure(x):
...     print("L'image par maFonction de", x, "est", maFonction(x))
...
>>> maProcédure(1)
L'image par maFonction de 1 est 0
>>> maProcédure(0)
L'image par maFonction de 0 est 1
>>> maProcédure(10)
L'image par maFonction de 10 est 81
```

Remarquer qu'une fonction peut être appelée dans la définition d'une autre fonction.

Remarquer que la fonction prédéfinie `print()` peut prendre plusieurs arguments séparés par des virgules, chaîne de caractère, variable numérique, etc...

Quelques exemples de variables de types :

- entier, (`nbr`)
- flottant, (`pi`)
- et chaîne de caractère (`str`).

```
Python 3.3.5 |Anaconda 2.0.1 (x86_64)| (default, Mar 10 2014, 11:22:25)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> # Exemples de VARIABLES de différents TYPES
...
>>> nbr = -12
>>> pi = 3.14159
>>> str = "Une chaîne de caractères"
>>> nbr
-12
>>> print(nbr, pi, str)
-12 3.14159 Une chaîne de caractères
>>> str
'Une chaîne de caractères'
>>> print(str)
Une chaîne de caractères
>>> █
```

Taper leur nom au prompt retourne leur valeur. L'affichage de la valeur est mieux formaté grâce à la fonction `print()`.

```
Python 3.3.5 |Anaconda 2.0.1 (x86_64)| (default, Mar 10 2014, 11:22:25)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> # Exemples de VARIABLES de différents TYPES
...
>>> nbr = -12
>>> pi = 3.14159
>>> str = "Une chaîne de caractères"
>>> nbr
-12
>>> print(nbr, pi, str)
-12 3.14159 Une chaîne de caractères
>>> str
'Une chaîne de caractères'
>>> print(str)
Une chaîne de caractères
>>> print("La circonférence d'un cercle de rayon 3 est",2*pi*3)
La circonférence d'un cercle de rayon 3 est 18.849539999999998
>>> █
```

On peut effectuer avec une variable de type entier, flottant, chaîne, (...) tout ce que l'on peut faire avec un type respectivement entier, flottant, chaîne, (...).

```
>>> # Affectation
... nbr = 7
>>> nbr
7
>>> nbr = 2 * nbr + 1
>>> nbr
15
>>> nbr = nbr ** 2
>>> nbr
225
>>> nbr = nbr + 1
>>> nbr
226
>>> nbr += 1
>>> nbr
227
>>> █
```

L'opération principale pour une variable est l'affectation représentée par le symbole =.

```
>>> # Affectation
... nbr = 7
>>> nbr
7
>>> nbr = 2 * nbr + 1
>>> nbr
15
>>> nbr = nbr ** 2
>>> nbr
225
>>> nbr = nbr + 1
>>> nbr
226
>>> nbr += 1
>>> nbr
227
```

Lors d'une affectation l'expression à droite du symbole = est évaluée, avant d'être affecté à la variable dont le nom figure à gauche du symbole =.

Le membre de gauche de = ne peut être que le nom d'une variable. Si elle n'existe pas encore la variable sera créée lors de l'affectation.

Si un même nom de variable figure des 2 côtés de =, la valeur de celle de droite est celle AVANT l'affectation, celle de gauche est celle APRES l'affectation.

L'affectation $x = x + 1$ n'a rien à voir avec l'équation mathématiques impossible $x = x + 1$.

```

>>> # Affectation
... nbr = 7
>>> nbr
7
>>> nbr = 2 * nbr + 1
>>> nbr
15
>>> nbr = nbr ** 2
>>> nbr
225
>>> nbr = nbr + 1
>>> nbr
226
>>> nbr += 1
>>> nbr
227

```

L'instruction :

$$\text{variable} \uparrow = \text{expression}$$

est équivalente à l'instruction :

$$\text{variable} = \text{variable} \uparrow \text{expression}$$

où $\uparrow$ désigne n'importe quelle opération : +, -, *, /, //, %, **, etc....

Une **variable** a :

- un **identifiant** : pour nous c'est son nom, qui permet de manipuler la variable au sein d'un programme ou d'une instruction (en mode console). C'est une chaîne de caractère alphanumérique, c.à.d. composée de lettres et de chiffres (et du symbole '_'), qui ne doit pas débiter par un chiffre. Eviter de le débiter par une lettre majuscule. Son nom doit être clair pour faciliter la relecture du programme.
- Un **type** : entier (relatif), flottant (réel...), complexe, chaîne de caractère, etc...
- Un **contenu**, c'est sa valeur. Elle est stockée dans la mémoire sous forme d'un nombre en écriture binaire (: qu'avec les chiffres 0,1).

En python la définition (déclaration) d'une variable se fait à l'aide d'une affectation : `variable = valeur` (voir des exemples plus haut).

```

>>> type(nbr)
<class 'int'>
>>> type(pi)
<class 'float'>
>>> type(str)
<class 'str'>

```

La fonction `type(.)` retourne le type d'une variable :

1. `int` pour un type entier,
2. `float` pour un type flottant,
3. `str` pour une chaîne de caractère, etc...

Python pratique le *typage dynamique* : il n'est pas besoin (au contraire d'autres langages) de déclarer le type d'une variable, python s'en charge.

```
>>> a = 1
>>> type(a), a
(<class 'int'>, 1)
>>> a = 1.0
>>> type(a), a
(<class 'float'>, 1.0)
```

Le type d'une variable peut être modifié... C'est cependant à déconseiller !

Une particularité de l'affectation de variables sous python

Python permet en une seule instruction d'affectation ('=') d'affecter plusieurs variables :

```
>>> # Affectations multiples :
... varnbr, varstr = 12, "bonjour"
>>> varnbr
12
>>> varstr
'bonjour'
>>> █
```

Les variables, à gauche de l'instruction '=' d'affectation, sont séparées par des virgules, de même que les valeurs, à droite de '=', qui doivent être en même nombre, et sont affectées de gauche à droite.

Exemple : Calcul des premiers termes de la suite de Fibonacci :

La suite Fibonacci est définie par :

$$u_0 = 0, u_1 = 1, \quad \forall n \in \mathbb{N}, \quad u_{n+2} = u_{n+1} + u_n$$

```
>>> # Calcul des premiers termes de la suite de Fibonacci
... u, v = 0, 1 ; print(u,v)
0 1
```

Remarquer l'utilisation du point-virgule pour séparer plusieurs instructions sur une même ligne !

```
>>> u, v = u+v, u+2*v ; print(u,v)
1 2
```

```
>>> u, v = u+v, u+2*v ; print(u,v)
3 5
>>> u, v = u+v, u+2*v ; print(u,v)
8 13
```

La touche ▲ du clavier (déplacement vers le haut) permet de relancer la ligne d'instructions précédentes. Elle permet plus généralement par des appuis répétés de relancer l'une quelconque des lignes précédentes.

Bien noter que durant une affectation multiple les valeurs (à droite du '=') sont celles avant l'appel de l'instruction. Ainsi l'instruction `a,b = b,a` échange les valeurs de 2 variables a et b :

```
>>> # Echange des valeurs de 2 variables
... a, b = 1, 2
>>> print(a,b)
1 2
>>> a,b = b,a
>>> print(a,b)
2 1
```

Comparer avec l'instruction suivante :

```
>>> a,b = b,a
>>> print(a,b)
2 1
>>> a = b = a
>>> print(a,b)
2 2
```

qui est équivalente à deux affectations simultanées : `b=a` suivie de `a=b` (l'affectation `a=b=a` est effectuée successivement de la droite vers la gauche), ou encore `b=a ; a=b`.

Dans d'autres langages pour échanger les valeurs de 2 variables il faut faire appel à une fonction prédéfinie (souvent `swap(. , .)`), ou procéder en plusieurs affectations.

A l'aide d'une variable temporaire :

```
>>> a, b = 1, 2
>>> vartemp = b ; b = a ; a = vartemp
>>> print(a,b)
2 1
```

Etat de la mémoire durant l'exécution :

1. `a, b = 1, 2`

a	Valeur initiale α (=1)
b	Valeur initiale β (=2)

2. `vartemp = b`

a	α (=1)
b	β (=2)
vartemp	β (=2)

3. `b = a`

a	α (=1)
b	α (=1)
vartemp	β (=2)

4. `a = vartemp`

a	β (=2)
b	α (=1)
vartemp	β (=2)

La deuxième ligne a permis d'échanger les valeurs contenues dans les variables `a` et `b`, grâce à une 3^{ème} variable `vartemp`.

Chapitre 2

Premiers programmes

2.1 Diverses utilisations de python

2.1.1 Ecriture d'un programme python

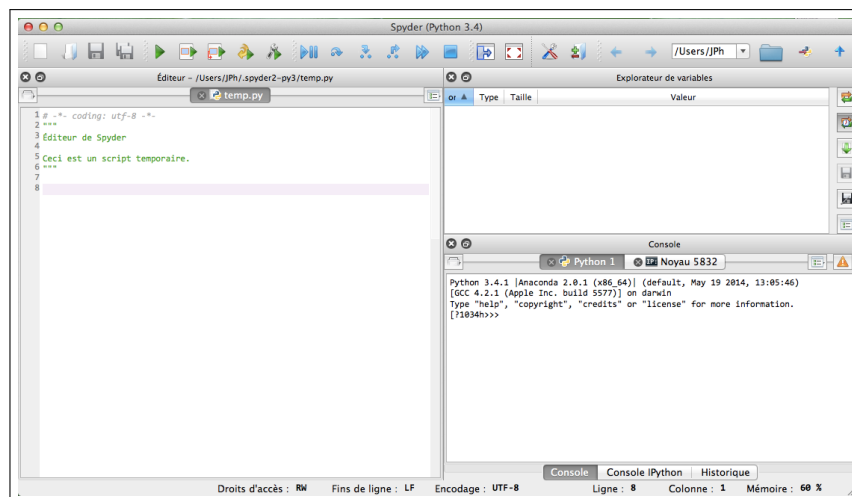
Ecrire un programme python

Le programme s'écrit dans un fichier texte que l'on sauvegarde avec l'extension `.py`. Le lancement du programme peut se faire à partir d'une console par la commande `python fichier.py`.

Il est préférable d'utiliser un environnement de développement ; il permet d'écrire des programmes, de les sauvegarder, modifier, etc..., de lancer leur exécution, de la stopper, et d'avoir une fenêtre interactive qui aide au développement du programme et retourne les résultats de l'exécution.

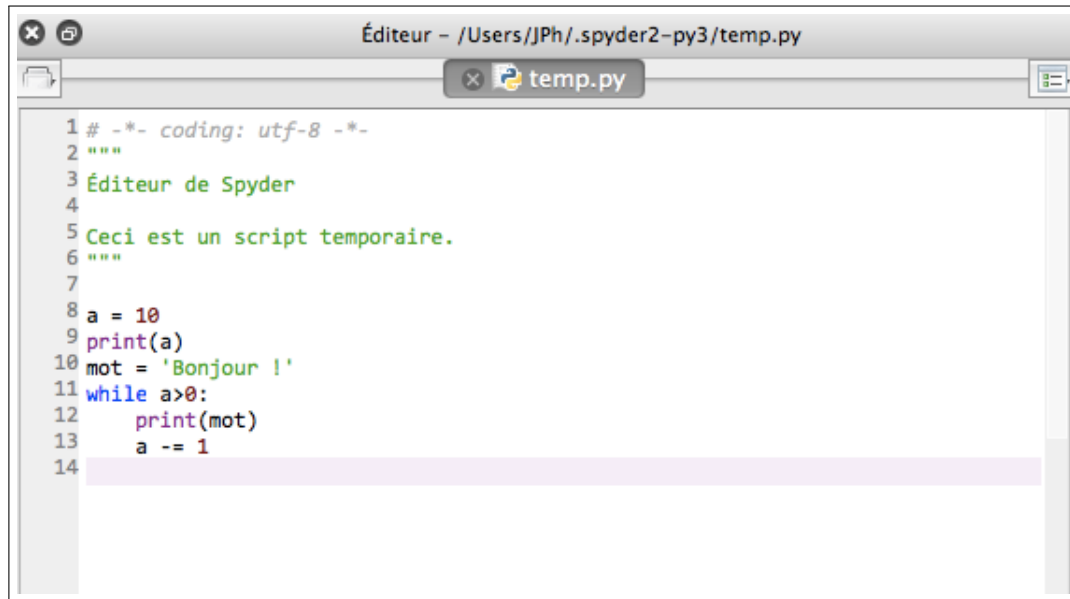
2.1.2 Environnement de programmation

Par exemple `anaconda`, à télécharger au lien suivant <http://continuum.io/downloads> avec la version 3.4 du langage :



Détails de l'environnement de programmation

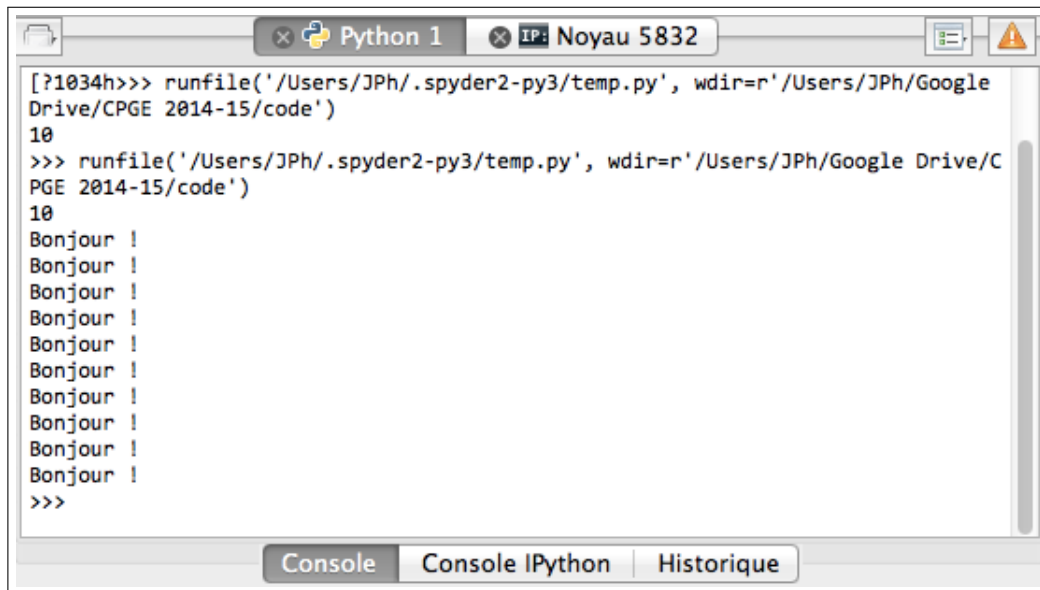
- Saisie du programme dans l'éditeur (à gauche).



The screenshot shows the Spyder Python IDE's editor window. The title bar reads 'Éditeur - /Users/JPh/.spyder2-py3/temp.py'. The code editor contains the following Python code:

```
1 # -*- coding: utf-8 -*-
2 """
3 Éditeur de Spyder
4
5 Ceci est un script temporaire.
6 """
7
8 a = 10
9 print(a)
10 mot = 'Bonjour !'
11 while a>0:
12     print(mot)
13     a -= 1
14
```

- La fenêtre 'console' en bas à droite fonctionne en mode interactif (python ou python).



The screenshot shows the Spyder Python IDE's console window. The title bar reads 'Python 1' and 'IP: Noyau 5832'. The console displays the output of the script execution:

```
[?1034h>>> runfile('/Users/JPh/.spyder2-py3/temp.py', wdir=r'/Users/JPh/Google
Drive/CPGE 2014-15/code')
10
>>> runfile('/Users/JPh/.spyder2-py3/temp.py', wdir=r'/Users/JPh/Google Drive/C
PGE 2014-15/code')
10
Bonjour !
Bonjour !
Bonjour !
Bonjour !
Bonjour !
Bonjour !
Bonjour !
Bonjour !
Bonjour !
Bonjour !
>>>
```

At the bottom of the console window, there are three tabs: 'Console', 'Console IPython', and 'Historique'.

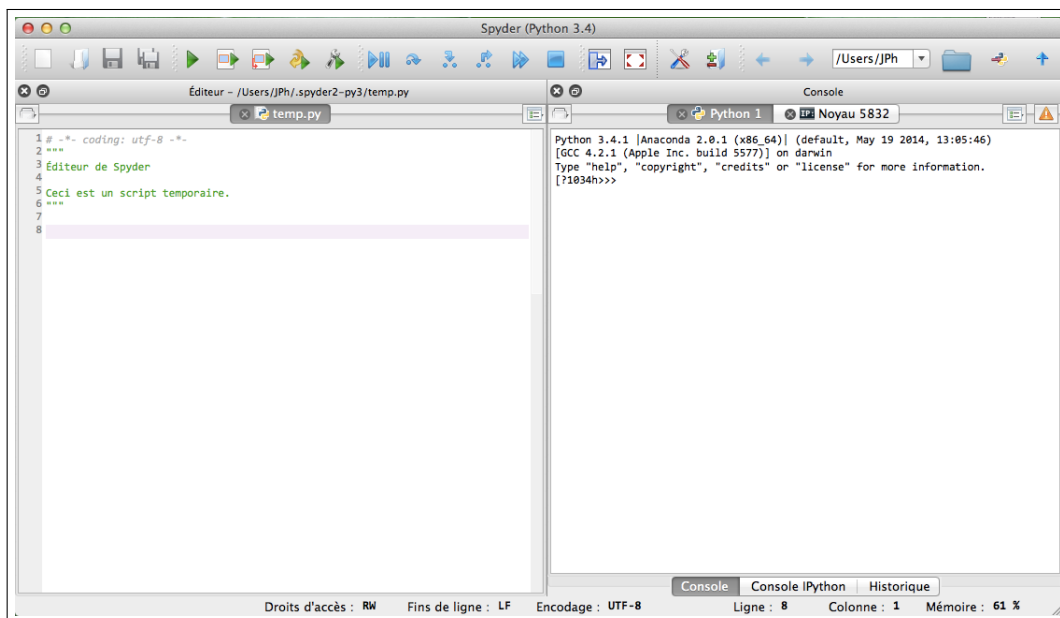
L'icône en haut à droite permet d'interrompre l'exécution d'un programme (qui bogue); celle à sa gauche de réinitialiser la console.

- La fenêtre 'explorateur de variable' (en haut à droite) donne l'état des variables après exécution.

Nom	Type	Taille	Valeur
a	int	1	0
mot	str	1	Bonjour !

Elle est surtout utile pour le débogage.

On peut aussi la fermer pour ne garder que les fenêtres essentielles : l'éditeur et une console.



2.2 Programmation en python

2.2.1 Saisie de données par l'utilisateur : input()

La fonction input()

Elle attend la saisie par l'utilisateur d'une chaîne de caractère, et retourne la valeur saisie. La chaîne de caractère doit être saisie sans apostrophes '.' ou guillemets ''.

```
>>> # La fonction input()
... chaine = input(); print("Vous avez saisi :",chaine)
Ceci est ma réponse
Vous avez saisi : Ceci est ma réponse
```

On peut passer à la fonction `input()` en argument une chaîne de caractère qui sera inscrite à l'écran.

```
>>> str = input('Saisissez votre réponse : '); print('Vous avez saisi :',str)
Saisissez votre réponse : Voici ma réponse !
Vous avez saisi : Voici ma réponse !
```

Si l'on n'a besoin d'une valeur numérique on convertit le résultat retourné à l'aide d'une fonction de conversion de type : `int()` ou `float`.

```
>>> n = int(input('Saisissez un nombre entier ')); print('Son carré est ',n**2)
Saisissez un nombre entier 3
Son carré est 9
```

Autrement un calcul produira erreur ou résultat inattendu.

```
>>> n = input('Saisissez un nombre entier '); print('Son carré est ',n**2)
Saisissez un nombre entier 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

- Attention en python 2 la fonction `input()` n'a pas le même comportement ; la fonction analogue y est `raw_input()` (voir l'annexe).
- Mon premier programme devient plus interactif :

```
>>> # Mon deuxième programme :
... r = input('Comment vous appelez-vous ? '); print('Bonjour',r)
Comment vous appelez-vous ? Toto
Bonjour Toto
```

L'utilisateur peut interagir avec le programme durant son exécution.

2.2.2 Un nouveau type de variables : booléen

Type booléen : bool

Lorsque l'on saisit les conditions suivantes les valeurs retournées peuvent prendre 2 valeurs : `True` ou `False`. Elles sont de type booléen 'bool' :

```
>>> 1 < 2
True
>>> 1 > 2
False
```

```
>>> a = (1<2)
>>> type(a)
<class 'bool'>
```

Opérateurs de comparaison :

<code>a < b</code>	a a une valeur strictement inférieure à celle de b
<code>a <= b</code>	a a une valeur inférieure ou égale à celle de b
<code>a == b</code>	a et b ont même valeur.
<code>a != b</code>	a et b ont des valeurs différentes.

et pareillement `a > b` et `a >= b`.

Opérations sur les booléens

On peut effectuer des opérations logiques sur les booléens (par ordre de priorité) : `or`, `and`, `not()` :

```
>>> a = (1 <= 2); b = (a == False); print(a,b)
True False
>>> a or b ; a and b ; not(a) ; not(b)
True
False
False
True
```

```
>>> var = "chaîne"; var == "Chaîne"; var != "Chaîne"; not(var == "Chaîne")
False
True
True
>>> (1 == 1.00) and ("chapeau" == 'chapeau')
True
```

La fonction suivante définit le connecteur logique `xor` (ou exclusif) :

```
>>> # Connecteurs logique xor : a xor b = a and not(b) or not(a) and b
... def xor(a,b):
...     return (a and not(b)) or (not(a) and b)
...
>>> xor(True,False), xor(False,True), xor(True,True), xor(False,False)
(True, True, False, False)
```

Conversion en booléen : `bool()`

La fonction de conversion `bool()` convertit une valeur de n'importe quel type en un booléen, selon les règles suivantes :

- Un nombre `'int'` ou `'float'` est converti à `True` s'il est non nul, à `False` sinon.
- Une chaîne de caractère `'str'` est convertie à `True` si elle est non vide, à `False` si c'est la chaîne vide : `""`.

```
>>> bool(3.14)
True
>>> bool(-7)
True
>>> bool('toto')
True
>>> bool('')
False
```

2.2.3 Structures de contrôle

Structures de contrôle

Pour l'instant nous avons vu que `python` est un langage de programmation impérative (un programme est une suite de ligne d'instructions), nous allons voir que c'est un langage de programmation structuré. Il reconnaît les structures de contrôles :

- une structure de test : `if ... [elif] ... [else]`
- deux structures de boucles : `while ...`
`for ...`

sans lesquelles les programmes s'exécuteraient séquentiellement ligne après ligne.

Nous allons commencer par voir la structure de boucle `while` puis la structure de test `if` (`elif`) (`else`). Nous étudierons plus tard la deuxième structure de boucle `for`.

`while` et `if` nous suffisent pour programmer. Tandis que `for` et `if` ne suffisent pas. Cependant `for` est particulièrement pratique, surtout en `python` !

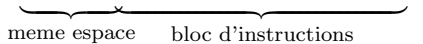
2.2.4 Structure de boucle while

L'instruction while

L'instruction `while` permet de répéter une séquence d'instruction, en boucle, tant qu'une *condition* est vérifiée.

```

        while condition:
.....  Instruction1
.....  Instruction2
.....  :
.....  Dernière instruction
    
```



En général *condition* s'évalue en un booléen, par exemple :

True, `a <= 0`, `(a == 0) and (b != -1)`.

```

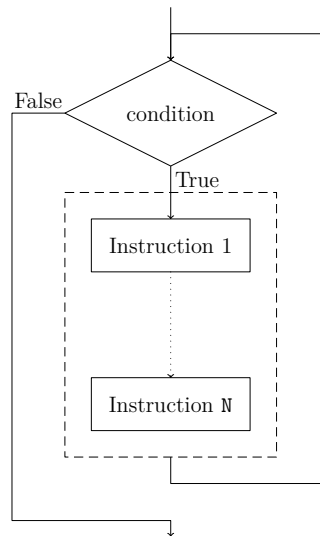
>>> x = 1e10    # Notation scientifique 1*10^10
>>> x
10000000000.0
>>> n = 0       # Initialisation de n : nombre de passages dans la boucle
>>> while (2**n < x):
...     n = n + 1
...
>>> print(n-1, ';', 2**(n-1), '<', x, '<=', 2**n)
33 ; 8589934592 < 10000000000.0 <= 17179869184
    
```

Ainsi $33 < \log_2(10^{10}) < 34$.

Exemple : pour attendre la saisie d'une chaîne non vide :

```
>>> # Pour attendre la saisie d'une chaîne non-vide
... var = ''
>>> while not(var):
...     var = input('Saisissez une chaîne non-vide ')
...
Saisissez une chaîne non-vide
Saisissez une chaîne non-vide
Saisissez une chaîne non-vide ok
>>> █
```

• Organigramme d'une boucle while :

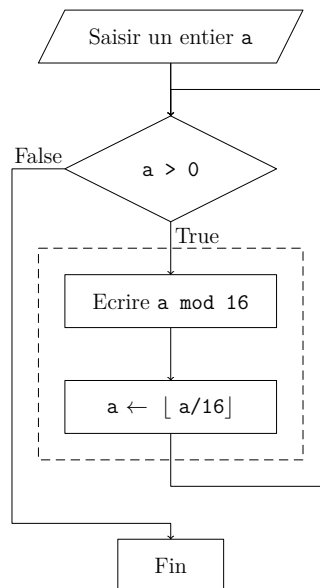


• On l'écrit en langage algorithmique :

Tant que condition faire :
 Instruction 1
 :
 Instruction N

• **Exemple : Ecrire un nombre en base 16. (1^{ère} ébauche.)**

<u>Code python :</u>	<u>En langage algorithmique :</u>
<code>a = int(input('Saisissez un entier'))</code>	Saisir un entier a
<code>while a > 0 :</code>	Tant que a > 0 faire :
<code>print a % 16</code>	écrire a modulo 16
<code>a = a // 16</code>	$a \leftarrow \lfloor a / 16 \rfloor$



- Pour comprendre son fonctionnement, modifions le code et regardons son exécution :

```

passage = 0
a = int(input('Saisissez un entier'))
while a > 0 :
    passage = passage + 1
    print('Passage ', passage)
    print('a modulo 16 : ', a % 16)
    a = a // 16
    print('a devient a//16 : ', a)
print('Sortie de la boucle')

```

```

Saisissez un entier : 72
Passage 1
a modulo 16 : 8
a devient a//16 : 4
Passage 2
a modulo 16 : 4
a devient a//16 : 0
Sortie de la boucle

```

Lorsque `not(a > 0)` la boucle `while` s'arrête.

- Amélioration du code (2^{ème} ébauche) :

```

a = int(input('Saisissez un entier'))
resultat = ''
while a > 0:
    resultat = str(a % 16) + ' ' + resultat
    a = a // 16
print(resultat)

```

L'exécution produit :

```

Saisissez un nombre 72
4 8

```

```

Saisissez un nombre 16 ** 3 + 15 * 16 ** 2 + 7
1 15 0 7

```

`str()` retourne la valeur de son paramètre convertie en chaîne de caractère.

Pour les chaînes de caractère '+' est l'opération de *concaténation* :

'aaaa' + 'bbbb' s'évalue en la chaîne 'aaaabbbb'.

Boucle sans fin

Attention une boucle peut ne jamais se terminer, et provoquer une exécution sans fin :

```

while True :
    print('bonjour')

```

provoque une affichage sans fin...

Cela peut être volontaire pour lancer une procédure indéfiniment : la procédure doit permettre l'arrêt :

```

while True:
    menu()

```

Ici `menu()` est une procédure (à écrire) qui proposera plusieurs choix à l'utilisateur. Il doit pouvoir choisir l'arrêt de l'exécution du programme.

Il est important de justifier de l'arrêt d'une boucle (à l'aide d'un *invariant de boucle*).

2.2.5 Structure de test

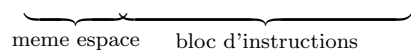
structure de test

```

if condition :
    .... Bloc d'Instructions

else :
    .... Bloc d'instructions

```



La condition est évaluée en booléen.

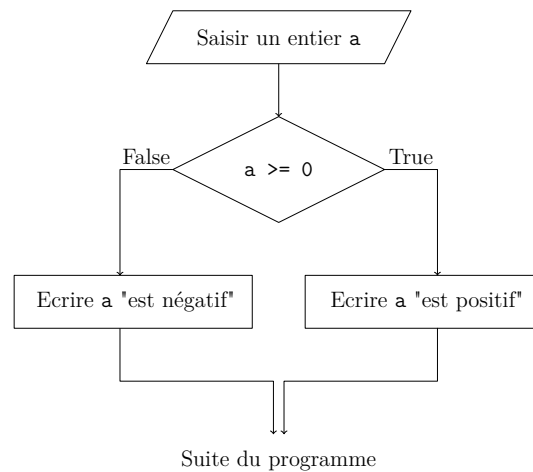
– Si elle a valeur `True` le premier bloc d'instruction est exécuté, le deuxième ne l'est pas, puis l'exécution du programme se poursuit.

– Si elle a valeur `False` le deuxième bloc d'instruction est exécuté (le premier ne l'est pas), puis l'exécution du programme se poursuit.

L'instruction `else` est optionnelle.

Structure de test if [else]

Code python :	En langage algorithmique :
<code>a = float(input('Saisissez un nombre'))</code>	Saisir un nombre a
<code>if a >= 0 :</code>	Si a >= 0 faire :
<code>print(a, 'est positif')</code>	écrire a, "est positif"
<code>else :</code>	Sinon faire :
<code>print(a, 'est négatif')</code>	écrire a, "est négatif"



Structure de test if [else]

- Exemple 1 : Une fonction pour déterminer si un nombre entier est pair ou impair :

```

def pair(n):
    if n%2 == 0:
        return True
    else:
        return False
  
```

- Exemple 2 : Dans le calendrier grégorien (actuel), une année est bissextile si soit :
 - Elle est divisible par 4 sans être divisible par 100.

- Elle est divisible par 400.

Ecrire une fonction qui décide si une année est ou non bissextile.

```
def bissextile(n):
    if n%400 == 0:
        return True
    if n%4 == 0:
        if n%100 == 0:
            return False
        else:
            return True
    else:
        return False
```

Remarque : une commande `return` provoque la sortie de la fonction.

Si dans le dernier programme on remplace les commandes `return` par `print()` (ce n'est plus une fonction mais une procédure), le programme bogue :

```
def bissextile(n):
    if n%400 == 0:
        print(n,'est une année bissextile')
    if n%4 == 0:
        if n%100 == 0:
            print(n,"n'est pas une année bissextile")
        else:
            print(n,'est une année bissextile')
    else:
        print(n,"n'est pas une année bissextile")
```

```
>>> bissextile(2000)
2000 est une année bissextile
2000 n'est pas une année bissextile
```

En effet la condition `n%400 == 0` est vérifiée et produit l'affichage de la première ligne, mais les conditions suivantes `n%4 == 0` et `n%100 == 0` sont aussi vérifiées et produisent l'affichage de la seconde ligne.

La solution consiste à imbriquer ces deux tests en conditionnant le second à l'échec du premier, de la façon suivante :

```
def bissextile(n):
    if n%400 == 0:
        print(n,'est une année bissextile')
    else:
        if n%4 == 0:
            if n%100 == 0:
                print(n,"n'est pas une année bissextile")
            else:
                print(n,'est une année bissextile')
        else:
            print(n,"n'est pas une année bissextile")
```

Le programme ne bogue plus :

```
>>> bissextile(2000)
2000 est une année bissextile
```

Structure de test if [elif] [else]

L'écriture de tests imbriqués :

```
if (condition1) :
    Bloc d'instructions 1
else :
    if (condition2) :
        Bloc d'instructions 2
    else :
        Bloc d'instructions 3
```

s'écrit à l'aide d'une seule structure de test :

```
if (condition1) :
    Bloc d'instructions 1
elif (condition2) :
    Bloc d'instructions 2
else :
    Bloc d'instructions 3
```

`elif` et `else` sont optionnels. `elif` peut être utilisé plusieurs fois dans une structure de test : `if ... elif ... elif ... else`.

Reprenons l'exemple de la fonction `bissextile`. En voici une version plus concise et lisible :

```
def bissextile(n):
    if n%400 == 0:
        print(n,'est une année bissextile')
    elif n%4 == 0 and n%100 != 0:
        print(n,'est une année bissextile')
    else:
        print(n,"n'est pas une année bissextile")
```

Le programme ne bogue plus :

```
>>> bissextile(2000)  
2000 est une année bissextile
```

Chapitre 3

Listes et boucle for

3.1 Les listes en python

3.1.1 Les listes

Les listes

- Dès que l'on commence à manipuler en python un grand nombre de données l'usage de variable numérique devient insuffisant.

Exemple : imaginons que l'on veuille stocker les notes des élèves d'une classe pour calculer leur moyenne et leur écart-type. On peut imaginer d'utiliser N variables de type `float` (N étant le nombre d'élèves de la classe), puis d'écrire des fonctions `moyenne(.)` et `ecarttype(.)` prenant N paramètres ; fastidieux lorsque N=47, et il faudra la réécrire pour chaque nouvel effectif...

```
def moyenne47(n1, n2, ..., n47):  
    return (n1 + n2 + ... + n47) / 47
```

- En python la structure de donnée qui va nous aider est la *liste*. c'est un **objet** de type `list` qui permet de collecter des éléments : données de type quelconque : `int`, `float`, `str`, `bool`, ou d'autres listes, etc...

Les listes

Exemple :

```
>>> liste = [1, 2, 3, 'toto']
```

```
>>> print(liste)  
[1, 2, 3, 'toto']  
>>> type(liste)  
<class 'list'>
```

```
>>> len(liste)  
4
```

```
>>> liste[0], liste[1], liste[len(liste) - 1]
(1, 2, 'toto')
```

Une liste est créée à l'aide d'une affectation. Ses éléments sont entre crochets `[.]`. La fonction `len(.)` prend en argument une liste et retourne son nombre d'éléments. Les éléments d'une liste s'obtiennent grâce à leur **indice** entre crochets. Attention le premier élément a pour indice 0 !

Les listes

Exemple : reprenons notre problème de calcul de la moyenne de notes. Mettons à profit ce que nous avons vu sur les listes ainsi que la fonction `sum(.)` qui prend en argument une liste et retourne, lorsque c'est possible, le résultat de l'opération `'+'` sur ses éléments.

```
>>> l = [10, 12, 14, 6, 8, 15, 3, 17]    # la liste de notes

>>> def moyenne(liste):                  # fonction moyenne(.)
...     return sum(liste) / len(liste)
...

>>> moyenne(l)
10.625
```

Saisissons la liste des notes. Définition de la fonction `moyenne(.)` qui s'applique à toute liste non vide de nombres. On obtient le résultat attendu, la moyenne est de 10.625.

Les listes sont des objets **modifiables** (on dit aussi **mutables**) : on peut modifier leurs éléments.

```
>>> liste = [1, 2, 3, 'toto']           # une liste

>>> liste[0] = 'le début'; liste[2] = [-1, -2, -3]
>>> print(liste)
['le début', 2, [-1, -2, -3], 'toto']

>>> print(liste[-1], liste[-2])
'toto' [-1, -2, -3]

>>> print(liste[-5], liste[4])
... IndexError: list index out of range
```

On modifie un élément de la liste en lui affectant une nouvelle valeur (de n'importe quel type, simple ou complexe). L'indice `-1` permet d'obtenir le dernier élément. C'est plus simple que `liste[len(liste) - 1]`. L'indice `-2` l'avant-dernier, etc... Un indice qui n'est pas compris entre `-len(liste)` et `len(liste)-1` produit une erreur `'IndexError'`.

```
>>> print(liste)
['le début', 2, [-1, -2, -3], 'toto']

>>> type(liste[0]), type(liste[1]), type(liste[2])
(<class 'str'>, <class 'int'>, <class 'list'>)
```

```
>>> print(liste[2][0])          # liste[2] est une liste
-1

>>> l = [['a', 'b'], ['c', 'd']]
>>> print(l[0][0], l[0][1], '\n', l[1][0], l[1][1])
a b
c d
```

Les éléments d'une liste sont des valeurs de différents types, celles qu'on lui a affectées.

Une liste de listes permet de constituer un tableau bi-dimensionnel, etc...

Appartenance d'un élément à une liste : in

- La commande `in` permet de déterminer si un élément appartient ou non à une liste.

```
>>> liste = [1, -3, 5, 17.0, 2]
>>> 1 in liste
True
>>> -1 in liste
False
>>> -3.0 in liste
True
>>> 17 in liste
False
>>> 'toto' in liste
False
```

Parcours d'une liste : for Variable in Liste:

- Avec en plus la commande `for` on peut faire parcourir à une variable les éléments d'une liste :

```
>>> for i in liste:
...     print(i)
...
1
-3
5
17.0
2
```

- Par exemple pour répéter 3 fois une instruction :

```
>>> liste = [0, 1, 2]
>>> for i in liste:
...     print('Bonjour')
...
Bonjour
Bonjour
Bonjour
```

3.1.2 Création d'un itérateur avec range()

La fonction range()

La dernière application du parcours d'une liste pour répéter une séquence d'instructions est intéressante. Mais couteuse en mémoire puisqu'une liste stocke en mémoire toutes les valeurs qu'elle contient.

- Pour cette raison (depuis la version 3 de python), la fonction `range(n)` retourne un *itérateur* sur les `n` premiers entiers naturels (de 0 à `n-1` inclus). Les valeurs intermédiaires ne sont plus stockées en mémoire. Cependant la commande `in` permet encore de déterminer l'appartenance d'un élément :

```
>>> print(range(10))
range(0,10)
>>> 2 in range(10)
True
>>> 10 in range(10)
False
```

- On peut convertir le résultat retourné en une liste grâce à la fonction de conversion `list()` :

```
>>> l = list(range(10)); print(l)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- Avec deux arguments entiers `m, n`, `range(m, n)` retourne un itérateur sur les $\max(0, n-m)$ entiers consécutifs qui sont $m \leq . < n$.

```
>>> print(list(range(0,10))); print(list(range(-1,11)))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[-1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 10]
```

- Avec trois arguments entiers `m, n, k`, `range(m, n, k)` retourne un itérateur sur la liste des entiers de la forme $m+k*p$ avec `p` entier naturel qui sont compris entre `m` (inclu) et `n` (exclu).

```
>>> print(list(range(0,10,2))); print(list(range(10,0,-2)))
[0, 2, 4, 6, 8]
[10, 8, 6, 4, 2]
```

Ainsi : `range(n)` est équivalent à `range(0,n)` et à `range(0,n,1)`.

```
>>> list(range(0,10,-1))    # produit la liste vide
[]
```

Par contre un argument non entier provoque un message d'erreur 'TypeError'.

Attention : Dans python 2, la fonction `range()` retourne non pas un itérateur mais une liste ; c'est la fonction `xrange()` qui retourne un intégrateur (cf. annexe).

3.2 la boucle for

3.2.1 Boucle for

Boucle for

L'instruction `for` permet de répéter une séquence d'instruction, en boucle, pour une variable qui décrit une liste (plus généralement un conteneur...) ou un itérateur d'éléments.

`for variable in liste/itérateur:`

```
..... Instruction1
..... Instruction2
..... :
..... Dernière instruction
```

 meme espace bloc d'instructions

On l'écrit en langage algorithmique, dans le cas d'un conteneur :

Pour tout i dans *liste* faire :

```
Instruction 1
:
Instruction N
```

Bien sûr la variable `i` peut apparaître dans le bloc d'instruction (comme *variable locale* : c'est une 'copie', la modifier n'affecte pas la liste).

On l'écrit en langage algorithmique, dans le cas d'un itérateur :

Pour tout i de 0 à N-1 faire :

```
Instruction 1
:
Instruction N
```

Le pas d'incrémentation peut être différent de 1 :

`for variable in range(n,m,k):`

```
..... Instruction1
..... Instruction2
..... :
..... Dernière instruction
```

 meme espace bloc d'instructions

On l'écrit en langage algorithmique :

Pour tout i de n à $m \pm 1$ par pas de k faire :
 Instruction 1
 $\vdots$
 Instruction N

Le signe $+$ ou $-$ dépend respectivement de $k > 0$ ou $k < 0$.

Boucles for et while

La boucle for :

```
for i in liste:
..... instruction 1
..... :
..... instruction N
```

produit un résultat identique à la boucle while :

```
L = len(liste)      # L est la longueur de liste
j = 0               # j est l'indice
while (j < L):      # tant que l'indice ne dépasse pas
..... i = liste[j]  # i est l'élément d'indice j
..... instruction 1
..... :
..... instruction N
..... j = j + 1     # incrémenter l'indice j
```

seulement lorsque `instruction1, ..., instruction N` ne modifient pas `liste`.

Une boucle `while` est plus générale. On a le droit de modifier la longueur de la liste dans la boucle `while` ; on n'en a pas le droit dans la boucle `for` (on peut toujours modifier les éléments de la liste). Dans une boucle `for` le nombre de répétition de la boucle est fixé à l'entrée dans la boucle ; c'est `len(liste)`.

3.2.2 Exemples

Exemple 1 : la liste des carrés d'entiers compris entre 0 et 20

Nous souhaiterions créer la liste des carrés des entiers compris entre 0 et 20.

Pour cela on utilisera la méthode `list.append()` appliquée aux objets de type liste qui permet d'ajouter un élément en queue de liste :

```
>>> liste=[ ]; print(liste)
[]
>>> liste.append('toto'); print(liste)
['toto']
>>> liste.append('le héros'); print(liste)
['toto', 'le héros']
```

Solution :

```
>>> listeCarrés = [ ]          # initialisation
>>> for i in range(21):
...     listeCarrés.append(i ** 2)    # actualisation
...
```

```
>>> print listeCarrés
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256,
289, 324, 361, 400]
```

Exemple 2 : premiers termes de la suite Fibonacci

Ecrire une fonction `fibonacci()` qui prend en argument un entier N et retourne une liste contenant les N premiers termes de la suite de Fibonacci : $u_0 = 0, u_1 = 1, \forall n \in \mathbb{N}, u_{n+2} = u_{n+1} + u_n$.

Solution :

```
>>> def fibonacci(N):
...     result = [0, 1]          # Initialisation
...     for k in range(2, N):
...         result.append(result[k - 1] + result[k - 2])
...     return result
```

La définition de la fonction est proche de la définition par récurrence de la suite, la boucle `for` jouant le rôle de la relation de récurrence.

```
>>> fibonacci(16)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]
```

Exemple 3 : Méthode des trapèzes pour le calcul approché d'intégrales

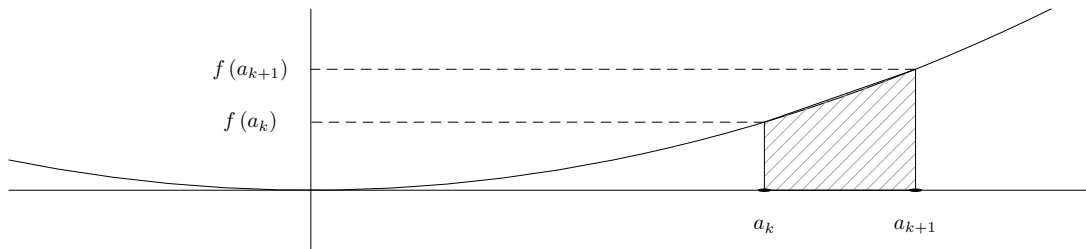
Soit l'application $f : x \mapsto x^2$ définie sur $\mathbb{R}$.

Ecrire une fonction `trapeze(.,.)` qui prend en argument deux réels a et b et retourne une approximation de l'intégrale de f de a à b par la méthode des trapèzes. Comparer son résultat avec la valeur quasi-exacte.

Méthode des trapèzes : si $f : [a, b] \rightarrow \mathbb{R}$ est continue :

Soit $a_k = a + k \frac{b-a}{n}$ pour $k \in [[0, n]]$ (ainsi $a_0 = a$, $a_n = b$). Si n est assez grand :

$$\begin{aligned} \int_a^b f(x) dx &\approx \left(\frac{b-a}{n} \right) \sum_{k=0}^{n-1} \frac{f(a_k) + f(a_{k+1})}{2} \\ &\approx \left(\frac{b-a}{n} \right) \left(\frac{f(a) + f(b)}{2} + \sum_{k=1}^{n-1} f(a_k) \right) \end{aligned}$$



```
def trapeze(a,b):
    if a > b: return -trapeze(b,a)
    N = 100
    pas = (b-a)/N
    res = (a**2+b**2)/2
    for k in range(1,N):
        res += (a + k*pas)**2
    res *= pas
    return res
```

la boucle for permet de calculer la somme.

```
>>> trapeze(0,1)      # intégrale de 0 à 1
0.33335
>>> 1**3 /3.         # valeur exacte
0.3333333333333333
>>> trapeze(0,100)    # intégrale de 0 à 100
333350.0
>>> 100**3 /3.        # valeur exacte
333333.3333333333
```

Amélioration de la précision du dernier exemple

```
def trapeze(a,b):          # Version 2, meilleure précision
    if a > b: return -trapeze(b,a)
    N = int(100*(b-a))
    pas = (b-a)/N
    res = 0
    for k in range(1,N):
        res += (a + k*pas)**2
    res *= pas
    return res
```

la précision est meilleure :

```
>>> print('approchée:',trapeze(0,1), 'exacte:', 1**3/3)
approchée: 0.33335 exacte: 0.333333333333
>>> print('approchée:',trapeze(0,100), 'exacte:', 100**3/3)
approchée: 333333.335 exacte: 333333.333333
```


Chapitre 4

Listes et autres conteneurs

4.1 Approfondissements sur les listes

4.1.1 Les méthodes de la classe list

Méthodes de la classe list

Méthode :	Action :
<code>liste.append(x)</code>	Pour ajouter <code>x</code> à la fin de la liste <code>liste</code>
<code>liste.extend(liste2)</code>	Pour ajouter la liste <code>liste2</code> à la suite de la liste <code>liste</code>
<code>liste.insert(i,x)</code>	Pour insérer l'élément <code>x</code> en position <code>i</code> dans <code>liste</code>
<code>liste.pop()</code>	Pour retirer et renvoyer le dernier élément dans <code>liste</code>
<code>liste.pop(i)</code>	Pour retirer et renvoyer l'élément en position <code>i</code> dans <code>liste</code>
<code>liste.remove(x)</code>	Pour retirer la première occurrence de <code>x</code> dans <code>liste</code>
<code>liste.index(x)</code>	Renvoie la 1 ^{ère} position de <code>x</code> dans <code>liste</code> . Message d'erreur si aucune.
<code>liste.count(x)</code>	Renvoie le nombre d'occurrences de <code>x</code> dans <code>liste</code> .
<code>liste.sort()</code>	Trie la liste par ordre croissant.
<code>liste.reverse()</code>	Renverse l'ordre des éléments de la liste.

- Méthodes `append()`, `extend()`, `insert()`, `pop()` : pour ajouter ou retirer un élément d'une liste via son indice.
- Méthode `remove()` : pour retirer le premier élément d'une liste d'une valeur donnée.
- Méthode `index()` : pour chercher un élément dans une liste.
- Méthode `count()` : pour compter les éléments d'une liste.
- Méthodes `sort()`, `reverse()` : manipulation de liste : triage et renversement.

Noter aussi que `x in liste` retourne `True` ou `False` selon si `x` apparaît ou non dans `liste`.

4.1.2 Saucissonnage ou slicing

Saucissonnage ou slicing

```
>>> liste = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
```

```
>>> liste1 = liste[2:5]    # tranche (slice) de la liste
>>> print(liste1)
['c', 'd', 'e']
```

L'instruction `>>> liste1 = liste[2:5]` crée une nouvelle liste `liste1` dont les éléments sont ceux de `liste` allant de l'indice 2 (inclus : le 3ème élément) à l'indice 5 (exclus : jusqu'au 5ème élément, celui d'indice 4). On l'appelle une *tranche* (slice en anglais) de la liste.

Les indices entre crochets peuvent sortir de la plage d'indice de la liste :

```
>>> liste2 = liste[0:100]   # tranche des indices de 0 à 100
>>> print(liste2)
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
```

Un troisième paramètre définit un pas :

```
>>> liste3 = liste[6:2:-1]   # tranche des indices de 6 à 2 par pas de -1
>>> print(liste3)
['g', 'f', 'e', 'd']
```

Les ':' définissent les champs, les valeurs de ces derniers étant optionnelles.

LISTE[Index départ (inclus) : Indice arrivée (exclus) : Pas]

`python` crée la tranche en copiant l'élément de `LISTE` d'indice `Indice de départ`, puis tous les éléments obtenus en ajoutant successivement `Pas` à l'indice, tant qu'on ne dépasse pas `Indice d'arrivée`. **Par défaut : Pas = 1**

- si `Pas > 0` : **par défaut** indice départ = 0 ; indice arrivée = `len(LISTE)`
- si `Pas < 0` : **par défaut** indice départ = `len(LISTE)-1` ; indice arrivée = -1

```
>>> liste4 = liste[::2]      # 2 slicing par pas de 2
>>> print(liste4)
['a', 'c', 'e', 'g', 'i']
liste5 = liste[::-1]        # 2 slicing par pas de -1
print(liste5)
['i', 'h', 'g', 'f', 'e', 'd', 'c', 'b', 'a']
```

```
>>> liste5 = liste[3::2]     # départ de l'indice 3, par pas de 2
>>> print(liste5)
['d', 'f', 'h']
liste6 = liste[:3:-1]       # départ de la fin par pas de -1, arrêt avant
                             l'indice 3,
>>> print(liste6)
['i', 'h', 'g', 'f', 'e']
```

```
>>> liste7 = liste[:]       # Pour copier une liste
>>> print(liste7)
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
```

4.1.3 Copie de liste

Copie de liste

Il faut prendre garde à la façon dont `python` copie une liste... Illustrons cela sur un exemple :

```
>>> liste = ['a', 'b', 'c']    # définition d'une liste
>>> copie = liste              # puis copie de la liste
>>> print(copie)
['a', 'b', 'c']
```

Copie d'une liste : jusqu'ici tout va bien.

```
>>> liste[0] = 'modifié'      # Modifions un élément dans liste
>>> print(liste)
['modifié', 'b', 'c']
```

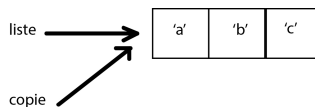
Modifions un élément de la liste originelle. Tout se passe comme prévu.

```
>>> print(copie)              # regardons ce qu'est devenue la copie
['modifié', 'b', 'c']
```

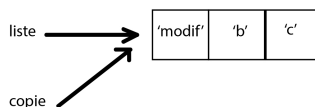
Mais, ô surprise, la copie aussi a été modifiée... Contrairement à ce qu'on aurait pu attendre.

Copie d'une liste : explication

En fait une liste en `python` ne contient que l'adresse mémoire où sont stockés ses éléments ; c'est ce que l'on appelle un pointeur.



Quand on copie `liste` dans `copie` c'est cette adresse mémoire qui est copiée. C'est un alias qui est créé. On parle de copie par référence.



Quand on modifie un élément d'une liste, il est alors aussi modifié dans la copie. L'avantage étant qu'on encombre moins la mémoire centrale puisque les éléments de la liste ne figurent qu'en un seul emplacement mémoire.

On peut contourner ce problème grâce à : `copie = liste[:]` qui fait une 'copie superficielle' :

```
>>> liste = ['a', 'b', 'c']      # définition d'une liste
>>> copie = liste[:]             # puis copie superficielle de la liste
>>> print(copie)
['a', 'b', 'c']
>>> liste[0] = 'modifié'        # Modifions un élément dans liste
>>> print(liste)
['modifié', 'b', 'c']
>>> print(copie)                # regardons ce qu'est devenue la copie
['a', 'b', 'c']
```

python fait une copie des éléments de la liste.. mais lorsqu'un élément est une liste, c'est l'adresse mémoire des objets de la liste qui est copiée. Aussi si l'un des éléments est une liste on retombe sur le même problème :

```
>>> liste = ['a', 'b', [0, 1]]   # définition d'une liste
>>> copie = liste[:]             # puis copie non-superficielle de la liste
>>> liste[0] = 'modifié'         # Modifions un élément dans liste
>>> liste[2][0] = 'MODIF'        # et un élément dans liste[2]
>>> print(liste); print(copie)   # regardons ce qu'est devenue la copie
['modifié', 'b', ['MODIF', 1]]
['a', 'b', ['MODIF', 1]]
```

Pour contourner totalement le problème utiliser la méthode `liste.deepcopy()` qui effectue une 'copie profonde'. Il faut avoir auparavant importé la bibliothèque `copy` :

```
>>> from copy import deepcopy; copie = deepcopy(liste).
```

4.1.4 Liste définie par compréhension

Compréhension de liste

On peut définir une liste à l'aide des mots-clés `for`, `in` et `if` comme on définit un ensemble en mathématiques 'par compréhension' :

$[f(x) \text{ for } x \text{ in liste}]$ correspond à $\{f(x) | x \in \text{liste}\}$

Exemple :

```
>>> liste = range(10)            # définition de la liste des entiers de 0 à 9
>>> LISTE = [x**2 for x in liste] # LISTE des carrés des éléments de liste
>>> print(LISTE)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

$[f(x) \text{ for } x \text{ in liste if Condition}(x)]$ correspond à $\{f(x) | x \in \text{liste tel que Condition}(x)\}$

Exemple :

```
>>> Liste = [x**2 for x in liste if (x%2==0)] # Carrés des éléments pairs
>>> print(Liste)
[0, 4, 16, 36, 64]
```

Exemple : Liste des multiples de 12 entre 0 et 100 :

```
>>> list = [x for x in range(101) if x % 12 == 0]    # Multiples de 12
>>> print(list)
[0, 12, 24, 36, 48, 60, 72, 84, 96]
```

Exemple : Liste des diviseurs d'un entier naturel N :

```
>>> N=120
>>> diviseurs = [x for x in range(1,N+1) if (N % x == 0)] # Diviseurs de N
>>> print(diviseurs)
[1, 2, 3, 4, 5, 6, 8, 10, 12, 15, 20, 24, 30, 40, 60, 120]
```

• **Exemple : application au calcul de la variance d'une série statistique.**

Une série statistique pourra être donnée par une liste de nombres.

• Fonction calculant l'espérance d'une série statistique :

```
def esperance(X):    # X est une liste numérique non vide
    return sum(X) / len(X)
```

On l'utilisera pour le calcul de la variance d'une série statistique :

• Calcul de la variance à l'aide de la définition : $V(X) = E((X - E(X))^2)$:

```
def variance(X):    # X est une liste numérique non vide
    m = esperance(X)
    Y = [ (x - m) ** 2 for x in X ]
    return esperance(Y)
```

• Calcul de la variance à l'aide de la formule de Koenig-Huygens : $V(X) = E(X^2) - E(X)^2$:

```
def variance(X):    # X est une liste numérique non vide
    X2 = [ x ** 2 for x in X ]
    return esperance(X2) - esperance(X)**2
```

4.2 Structures de données séquentielles

4.2.1 Opérations communes

Types séquentiels

Les types `int`, `float`, `bool` sont des *types scalaires*.

Les types `list` (listes) et `str` (chaînes de caractère) sont des structures de données *séquentielles*.

Tous les objets de type séquentiel ont en commun les opérations suivantes :

Opération	Résultat
<code>s[i]</code>	élément d'indice <code>i</code> de <code>s</code>
<code>s[i:j]</code>	Tranche de <code>i</code> (inclus) à <code>j</code> (exclus)
<code>s[i:j:k]</code>	Tranche de <code>i</code> à <code>j</code> par pas de <code>k</code>
<code>len(s)</code>	Longueur de <code>s</code>
<code>max(s), min(s)</code>	Plus grand et plus petit élément de <code>s</code>
<code>x in s</code>	<code>True</code> si <code>x</code> est dans <code>s</code> , <code>False</code> sinon
<code>x not in s</code>	<code>True</code> si <code>x</code> n'est pas dans <code>s</code> , <code>False</code> sinon
<code>s+t</code>	Concaténation de <code>s</code> et <code>t</code>
<code>s*n, n*s</code>	Concaténation de <code>n</code> copies de <code>s</code>
<code>s.index(x)</code>	Indice de la 1 ^{ère} occurrence de <code>x</code> dans <code>s</code>
<code>s.count(x)</code>	Nombre d'occurrences de <code>x</code> dans <code>s</code>

où `s` et `t` sont des objets séquentiels de même type, et `i`, `j`, `k`, `n` sont des entiers.

4.2.2 Les chaînes de caractère

les chaînes de caractères

Les objets de type `str`, chaînes de caractère, sont des structures de données séquentielles :

```
>>> chaine = 'Amanda'
>>> len(chaine)
6
>>> print(chaine[::-1])
adnamA
>>> chaine.count('a')
2
>>> print(chaine*2)
AmandaAmanda
>>> max(chaine)
'n'      # minuscules sont > majuscules puis ordre alphabétique
```

Ce ne sont pas des objets modifiables : changer un élément produit une erreur `TypeError` :

```
>>> chaine[0] = 'Z'
TypeError: 'str' object does not support item assignment
```

Exemple : Ecrire une fonction qui teste si une chaîne est un palindrome :

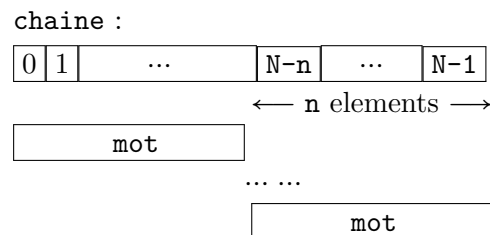
```
>>> def palindrome(c):
...     if (c == c[::-1]): return True
...     else: return False
```

Exemple : Recherche d'un mot dans une chaîne

La recherche d'un mot dans une chaîne de caractère est un problème essentiel en informatique qui admet des algorithmes élaborés et efficaces (temps d'exécution, utilisation de la mémoire).

Donner une solution naïve, pas très efficace n'est pas difficile :

```
def contient(chaine,mot):
    N = len(chaine)
    n = len(mot)
    for i in range(N-n+1):
        if (mot == chaine[i:i+n]): # comparaison du mot et d'une tranche
            return True
    return False
```



Le résultat est lent et coûteux en mémoire car la fonction doit effectuer jusqu'à $(N-n)$ copies d'une liste de longueur n , et pour chacune n comparaison, soit de l'ordre de $(N-n)*2n$ opérations et $(N-n)*n$ fois l'emplacement nécessaire au stockage d'un caractère utilisé.

4.2.3 les tuple ou sequence

les structures de données tuple

En français *t-uplet*. Ce sont des objets séquentiels.

On les définit par la liste de leurs éléments (comme pour une liste mais) entre parenthèses `(.)`.

```
>>> seq = (0,1,2,3)
>>> print(seq)
(0, 1, 2, 3)
>>> print(seq[0])
0
>>> print(seq*2)
(0, 1, 2, 3, 0, 1, 2, 3)
>>> seq[0] = 1
[...]
TypeError: 'tuple' object does not support item assignment
```

Ils diffèrent des listes surtout en ce que ce sont des objets non-modifiables (non-mutable) : `seq[0] = 1` produit une erreur `TypeError`.

Les parenthèses sont optionnelles ; nous les avons déjà utilisées par l'usage des virgules :

```
>>> a = 2
>>> 2*a, 3*a, 4*a    # retourne un t-uplet
(4, 6, 8)
```

4.3 Les dictionnaires

4.3.1 Les dictionnaires

Ce sont des structures de données qui ne sont pas séquentielles : un élément n'est plus repéré à l'aide d'un indice mais à l'aide d'un nom (sa clef). C'est un *champ* : couple d'une clef et de sa valeur. La valeur peut être de type quelconque, la clef un nombre une chaîne ou un t-uplet.

Cette structure de donnée s'appelle en python un *dictionnaire* et dans d'autres langages un *enregistrement*.

Un dictionnaire se définit entre accolades `{.}` par la suite des champs (la clef suivie de la valeur, séparés de `:`).

```
France = {'Capitale' : 'Paris', 'Superficie' : 674800, 'Population' : 65000000, 'Langue' : 'Français'}
```

C'est identique à :

```
France = {}
France['Capitale'] = 'Paris'
France['Superficie'] = 674800
France['Population'] = 65000000
France['Langue'] = 'Français'
```

4.3.2 Opérations sur les dictionnaires

L'accès à un élément se fait à l'aide de sa clef :

```
>>> France['Capitale']
'Paris'
```

Pour ajouter un champ affecter une valeur à une nouvelle clef. Pour supprimer un champ utiliser la fonction `del()`.

```
>>> France['Continent'] = 'Europe'    # ajout d'un champ
>>> del(France['Capitale'])            # suppression d'un champ
>>> print(France)
{'Continent' : 'Europe', 'Superficie' : 674800, 'Population' : 65000000, 'Langue' : 'Français'}
```

Les dictionnaires admettent les méthodes suivantes :

<code>dict.keys()</code>	retourne la liste des clefs du dictionnaire <code>dict</code>
<code>dict.values()</code>	retourne la liste des valeurs du dictionnaire <code>dict</code>
<code>dict.items()</code>	retourne la liste des champs (<code>key,value</code>) de <code>dict</code>
<code>dict.copy()</code>	retourne une copie du dictionnaire.

4.3.3 Exemple

Pour enregistrer une liste constituée d'un dictionnaire pour chaque élève d'une classe d'effectif 47 élèves, chaque dictionnaire comportant des champs pour les nom, prénom, age, 2eme langue :

```
eleves = []
for i in range(47):
    dict = {}
    dict['Nom'] = raw_input('Nom? ')
    dict['Prénom'] = raw_input('Prénom? ')
    dict['Age'] = int(raw_input('Age? '))
    dict['LV2'] = raw_input('Seconde Langue?')
    eleves.append(dict)
```

Remarque : le dictionnaire `dict` est une variable locale, une nouvelle est créée à chaque passage dans la boucle `for`, qui n'a plus rien à voir avec les définitions de `dict` précédentes.

Ecrivons maintenant une fonction qui recherche si un élève de nom donné est dans la classe et si oui retourne son dictionnaire.

```
def search(nom):
    for e in eleves:
        if e['Nom'] == nom:
            return e
    return False
```


Chapitre 5

Les modules

5.1 Utilisation d'un module

5.1.1 Importer un module

- Un module est un fichier ayant pour extension `.py` contenant des définitions de constantes et fonctions. Tout programmeur `python` peut réaliser un module. Importer un module permet d'utiliser ses constantes et fonctions.

Nous avons déjà utilisé les modules `math` et `random`.

- Pour importer un module :

1 Première méthode : A l'aide de l'instruction `from`

Pour importer une fonction ou constante :

```
>>> from math import sqrt
>>> sqrt(2)
1.4142135623730951
```

```
>>> from math import cos, pi
>>> cos(pi/4)
0.7071067811865476
```

Pour importer toute la bibliothèque :

```
>>> from math import *
>>> sin(pi/4)
0.7071067811865476
```

2 Deuxième méthode : Sans l'instruction `from`. Méthode conseillée.

```
>>> import math
>>> math.sqrt(2)
1.4142135623730951
>>> math.cos(math.pi/4)
0.7071067811865476
```

Fonctions et constantes doivent être précédées d'un préfixe : le nom du module suivi d'un

point '.'.

On peut définir un alias à l'aide de l'instruction `as` :

```
>>> import math as mth          # Création d'un alias mth
>>> # Utiliser alors le préfixe mth. devant un objet de math :
>>> mth.sqrt(2)
1.4142135623730951
>>> mth.cos(mth.pi/4)
0.7071067811865476
```

5.1.2 Exemple : le module `random`

Voici quelques fonctions fournies par le module `random` :

<code>randrange(a,b,k)</code>	Choisit un entier aléatoirement dans <code>range(a,b,k)</code>
<code>randint(a,b)</code>	Choisit un entier aléatoirement dans <code>[[a,b]]</code>
<code>choice(List)</code>	Choisit un <u>entier</u> aléatoirement dans la liste <code>List</code>
<code>random()</code>	Choisit un float aléatoirement dans <code>[0,1[</code>
<code>uniform(a,b)</code>	Choisit un float aléatoirement dans <code>[a,b[</code>

Ici aléatoirement signifie selon une loi uniforme (quasiment).

Exemple : Que fait le programme suivant ?

```
import random as rand
def de6(n):
    tirs = [0, 0, 0, 0, 0, 0]
    for i in range(n):
        t = rand.randint(1,6)
        tirs[t-1] += 1
    for j in range(6):
        tirs[j] = tirs[j]*100.0/n
        # 2 chiffres apres la virgule et symbole %:
        tirs[j] = '{:.2f}'.format(tirs[j]) + '%'
    return tirs
```

A l'aide de la méthode `format` pour le type `str`, l'expression `'{:.2f}'.format(x)`, pour un flottant `x`, retourne une chaîne de caractères représentant l'écriture décimale du nombre `x` avec 2 chiffres après la virgule.

Réponse : le programme simule `n` lancers d'un dé 6, et compte le nombre de fois où chaque face apparaît pour finalement retourner le pourcentage d'apparition de chaque face.

Maintenant vérifions le théorème des grands nombres : pour un grand nombre de tirs les fréquences d'apparition de chaque face devraient tendre vers leur probabilité de tir, ici donc être égales (la loi est uniforme : le dé est non pipé).

```

>>> de6(10)
['30.00%', '0.00%', '20.00%', '20.00%', '10.00%', '20.00%']
>>> de6(100)
['10.00%', '22.00%', '19.00%', '20.00%', '15.00%', '14.00%']
>>> de6(1000)
['16.00%', '15.40%', '16.10%', '16.90%', '18.50%', '17.10%']
>>> de6(10000)
['16.45%', '16.66%', '16.50%', '16.84%', '17.37%', '16.18%']
>>> de6(100000)
['16.57%', '16.67%', '16.80%', '16.62%', '16.66%', '16.68%']
>>> de6(1000000)
['16.65%', '16.69%', '16.66%', '16.64%', '16.63%', '16.74%']
>>> de6(10000000)
['16.66%', '16.66%', '16.65%', '16.66%', '16.69%', '16.68%']

```

La fréquence d'apparition de chaque face est pour un grand nombre de tir proche de sa probabilité $1/6$.

5.2 numpy et matplotlib

5.2.1 Modules scientifiques

Il existe de nombreux modules scientifiques sous `python` :

1. **numpy** : Puissant outil pour créer, manipuler, et appliquer de nombreuses opérations sur des tableaux de nombres (matrices). Stable et bien documenté :
<http://docs.scipy.org/doc/numpy/reference/> (en anglais).
2. **scipy** : Fonctions mathématiques puissantes s'appliquant aux tableaux générés par **numpy**. C'est la boîte à outil numérique pour les tableaux **numpy**. Elle contient des opérations spécifiques (algèbre linéaire, statistiques,...) de manipulation de tableaux, de plus haut niveau que celles de **numpy**. Documentation à :
<http://docs.scipy.org/doc/scipy/reference/> (en anglais).
3. **matplotlib** : Permet le tracé de graphes de fonctions.
http://matplotlib.org/users/pyplot_tutorial.html (en anglais).

Nous allons commencer par voir de plus près les modules **numpy** et **matplotlib**.

5.2.2 Le module numpy

- Le module **numpy** permet de créer, de manipuler, des tableaux de nombres, ou matrices, et de leur appliquer des opérations mathématiques courantes.

- **La fonction `array()`** permet de créer un tableau, ou **array**, à partir d'un tableau **python** c'est à dire d'une liste de listes de nombres (de même longueur) :

```
>>> import numpy as np
>>> A = np.array([[1,1,1],[0,1,1],[0,0,1]])
>>> A
array([[1, 1, 1],
       [0, 1, 1],
       [0, 0, 1]])
>>> print(A)
[[1 1 1]
 [0 1 1]
 [0 0 1]]
```

- La fonction `arange()` crée une matrice ligne de façon assez analogue à la fonction `range()`, à ceci près que les coefficients ne sont pas forcément entiers :

```
>>> v = np.arange(0, 1.5, 0.5)
>>> v
array([ 0., 0.5, 1. ])
>>> 2*v
array([ 0., 1., 2. ])
>>> -2*v + 10
array([ 10., 9., 8. ])
```

On peut appliquer sur les tableaux de nombreuses opérations arithmétiques, certaines étant comprises terme à terme.

- On accède aux éléments d'un tableau comme en `python`, grâce à un indice entre crochets :

```
>>> A[0], A[1]
(array([1, 1, 1]), array([0, 1, 1]))
>>> A[1][0]
0
```

- On peut appliquer du slicing :

```
>>> A[0:1]
array([[1, 1, 1]])
A[::-1]
array([[0, 0, 1],
       [0, 1, 1],
       [1, 1, 1]])
```

- Le type d'une matrice s'obtient grâce à la fonction `shape()`, son nombre d'élément grâce à `size()`. La fonction `reshape()` permet de changer la forme (=type) d'une matrice :

```
>>> L=np.arange(0,10)
>>> np.size(L)
10
>>> np.reshape(L,(2,5))
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> L
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> M = np.reshape(L,(2,5)) ; np.shape(M)
(2, 5)
```

- Le produit matriciel s'obtient à l'aide de la fonction `dot()`.

Un vecteur peut s'écrire à l'aide d'un tableau uni-dimensionnel.

Par exemple : `v= np.arange(0,3)` ou `v=array([0, 1, 2])` mais aussi comme la matrice colonne `v=array([[0],[1],[2]])`.

Exemple : avec la matrice 3×3 : $A = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$ et le vecteur $v = (0, 1, 2)$.

Le produit $A \times v$ est défini égal à $(3,3,2)$; le produit $v \times A$ n'est pas défini. Mais ${}^t v \times A = (0, 1, 3)$.

```
>>> v = array([0,1,2])
>>> np.dot(A,v)
array([3, 3, 2])
>>> np.dot(v,A)
array([0, 1, 3])
```

On le voit, lorsque la multiplication est impossible, pour $v.A$, la fonction `dot()` effectue $A \times {}^t v$, qui a retourné pour résultat $(0,1,3)$. On pourra saisir des matrices lignes sous forme de vecteurs (leur transposée). Ce fonctionnement est commun aux logiciels de calcul sur des matrices (`matlab`, `scilab`).

Pour effectuer le produit scalaire de 2 "vecteurs" utiliser la fonction `vdot()`.

- La fonction `transpose()` permet d'obtenir la transposée :

```
>>> np.transpose(A)
array([[1, 0, 0],
       [1, 1, 0],
       [1, 1, 1]])
```

Attention la "transposée" d'un "vecteur" est encore un "vecteur"...

- Attention l'opération `A ** 2` correspond à une élévation au carré terme à terme :

```
>>> A ** 2
array([[1 1 1]
       [0 1 1]
       [0 0 1]])
>>> (2*A) ** 2
array([[4 4 4]
       [0 4 4]
       [0 0 4]])
```

- Pour élever A au carré faire plutôt :

```
>>> np.dot(A,A)
array([[1, 2, 3],
       [0, 1, 2],
       [0, 0, 1]])
```

- Pour élever une matrice carrée A à une puissance n :

```
>>> B = A
>>> for i in range(1,n):
       B = np.dot(A,B)      # à la sortie de boucle B contient A^n
```

- De même l'inversion $A ** -1$ se fait terme à terme. Elle produira ici une erreur (division par 0) alors même que la matrice A est inversible.

- Certaines fonctions plus spécifiques sont disponibles dans un sous module. Pour l'algèbre linéaire c'est le sous-module `linalg` de `numpy`. Toutes les fonctions de ce sous-module devront être saisies avec le préfixe `np.linalg..`

- Pour l'inversion de matrice : utiliser la fonction `inv()` du sous-module `linalg` :

```
>> np.linalg.inv(A)
array([[ 1., -1.,  0.],
       [ 0.,  1., -1.],
       [ 0.,  0.,  1.]])
```

- le sous-module `linalg` contient aussi, entre autres :
 1. la fonction `det()` qui retourne le déterminant d'une matrice.
 2. La fonction `solve(A,b)` qui résout le système linéaire de matrice A et de second membre b (vecteur ou ligne).

Exemple : résoudre le système linéaire :

$$\begin{cases} x - y + z = 1 \\ -x + y + z = 1 \\ 2x - y - z = 0 \end{cases}$$

Solution :

```
>>> A=np.array([[1,-1,1],[-1,1,1],[2,-1,-1]])
>>> b= np.array([1,1,0])
>>> np.linalg.solve(A,b)      # avec solve()
array([ 1.,  1.,  1.])
>>> np.dot(np.linalg.inv(A),b)  # en inversant A
array([ 1.,  1.,  1.])
```

• **Principales fonctions du module numpy :**

<code>array(l)</code>	crée un tableau à partir d'une liste <code>l</code>
<code>arange(a,b,k)</code>	crée un vecteur dont les coefs sont les <code>a+k.N</code> entre <code>a</code> (inclu) et <code>b</code> (exclu).
<code>linspace(a,b,n)</code>	crée un vecteur de <code>n</code> valeurs régulièrement espacées entre <code>a</code> et <code>b</code> (inclus)
<code>zeros(p)</code>	crée un tableau de taille <code>p</code> rempli de zéros
<code>zeros((p,q))</code>	crée un tableau de taille <code>(p,q)</code> rempli de zéros
<code>ones(p)</code>	crée un tableau de taille <code>p</code> rempli de uns
<code>ones((p,q))</code>	crée un tableau de taille <code>(p,q)</code> rempli de uns
<code>shape()</code>	pour obtenir la taille d'un tableau (= type d'une matrice)
<code>size()</code>	pour obtenir le nombre d'éléments d'un tableau
<code>reshape()</code>	pour redimensionner un tableau
<code>dot()</code>	pour effectuer un produit matriciel de 2 matrices
<code>vdot()</code>	pour effectuer un produit scalaire de 2 "vecteurs"
<code>transpose()</code>	pour transposer une matrice
<code>rank()</code>	rang d'une matrice
<code>mean()</code>	valeur moyenne d'un tableau
<code>odeint()</code>	pour intégrer une équation différentielle...
<code>linalg :</code>	
<code>inv()</code>	inversion d'une matrice
<code>det()</code>	déterminant d'une matrice
<code>solve(A,b)</code>	résolution du système linéaire $A.X = b$

- numpy contient aussi les constantes et fonctions mathématiques usuelles.

5.2.3 Le module matplotlib

Pour le simple tracé de courbes nous n'utiliserons que le sous-module `pyplot`, importé, avec alias, à l'aide de la commande :

```
>>> import matplotlib.pyplot as plt
```

cf. documentation à : <http://www.matplotlib.org>.

- Les fonctions essentielles de `pyplot` sont :
 1. `plot()` pour le tracé de points, de courbes, et
 2. `show()` pour afficher le graphique créé.

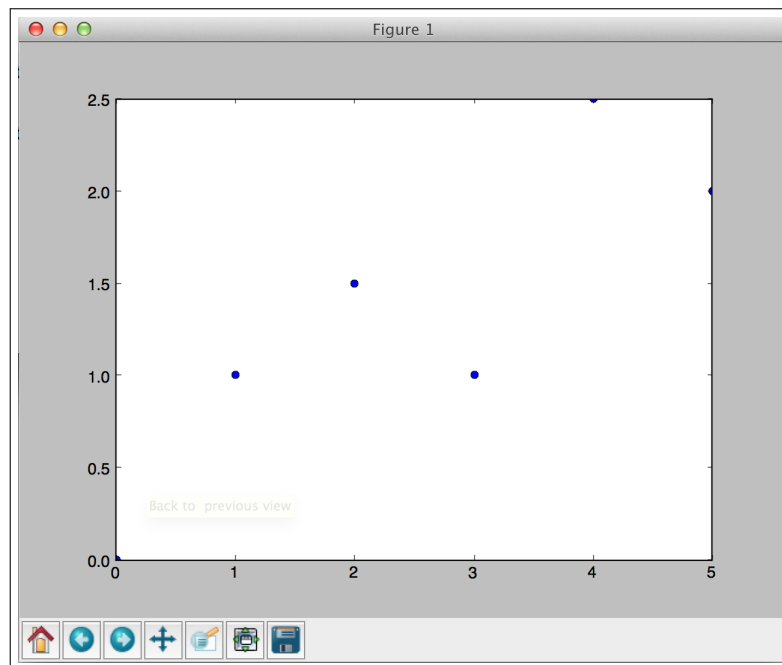
- Utiliser `plot()` avec :

1. en 1^{er} argument la liste des abscisses,
2. en 2^{eme} argument la liste des ordonnées,
3. en 3^{eme} argument (optionnel) le motif des points :
 - (a) `'.'` pour un petit point,
 - (b) `'o'` pour un gros point,
 - (c) `'+'` pour une croix,
 - (d) `'*'` pour une étoile,
 - (e) `'-'` points reliés par des segments
 - (f) `'--'` points reliés par des segments en pointillés
 - (g) `'-o'` gros points reliés par des segments (on peut combiner les options)
 - (h) `'b'`, `'r'`, `'g'`, `'y'` pour de la couleur (bleu, rouge, vert, jaune, etc...)
 - (i) cf. http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.plot.

- Exemple : pour le tracé d'un nuage de points :

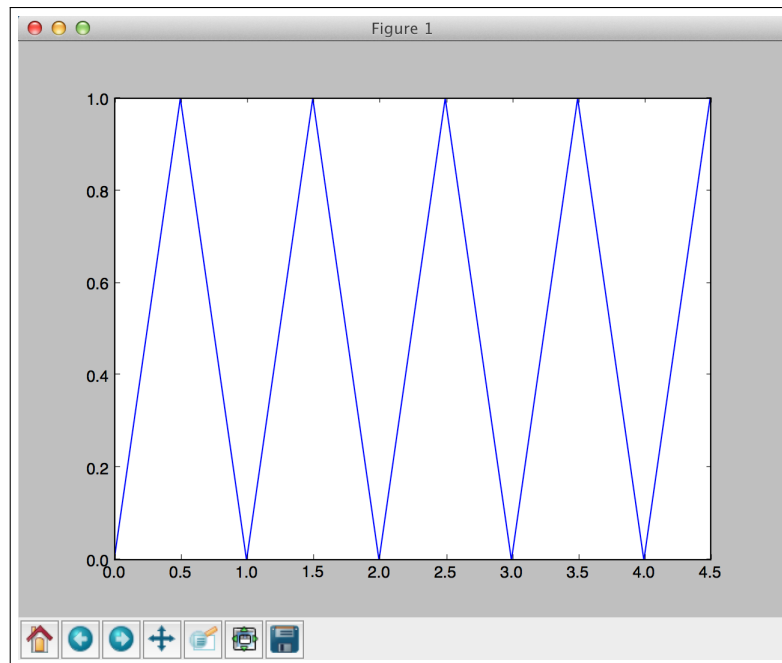
```
>>> import matplotlib.pyplot as plt
>>> abs = [0, 1, 2, 3, 4, 5]
>>> ord = [0, 1, 1.5, 1, 2.5, 2]
>>> plt.plot(abs, ord, 'o')
[<matplotlib.lines.Line2D object at 0x10c6610d0>]
>>> plt.show()
```

produit un graphique (au format .png) :



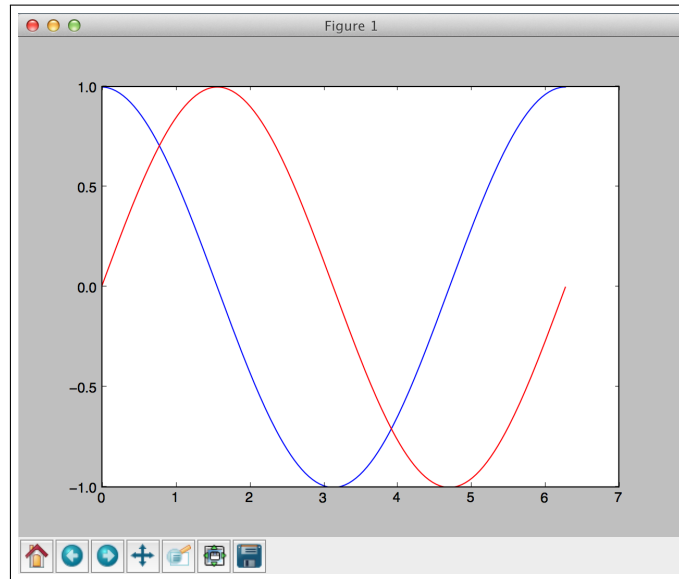
- Exemple : pour le tracé d'une ligne brisée :

```
>>> import matplotlib.pyplot as plt
>>> abs = [n/2. for n in range(10)]
>>> ord = [n % 2 for n in range(10)]
>>> plt.plot(abs,ord,'-b')
[<matplotlib.lines.Line2D object at 0x10dd1fa10>]
>>> plt.show()
```



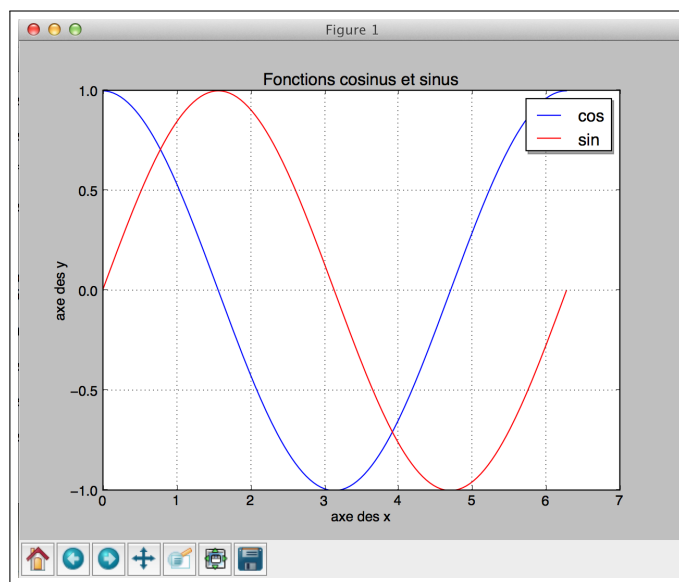
- Exemple : pour le tracé de courbes représentatives de fonctions réelles :

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np      # pour linspace() et les fonctions mathématiques
>>> X = np.linspace(0, 2*np.pi, 256) # X = 256 pts régulièrement espacés
>>> Ycos = np.cos(X)         # image directe de X par cos
>>> Ysin = np.sin(X)         # image directe de X par sin
>>> plt.plot(X,Ycos,'b')     # tracé de la courbe de cos en bleu
[<matplotlib.lines.Line2D object at 0x10d5e2b50>]
>>> plt.plot(X,Ysin,'r')     # tracé de la courbe de sin en rouge
[<matplotlib.lines.Line2D object at 0x1073aad90>]
>>> plt.show()
```



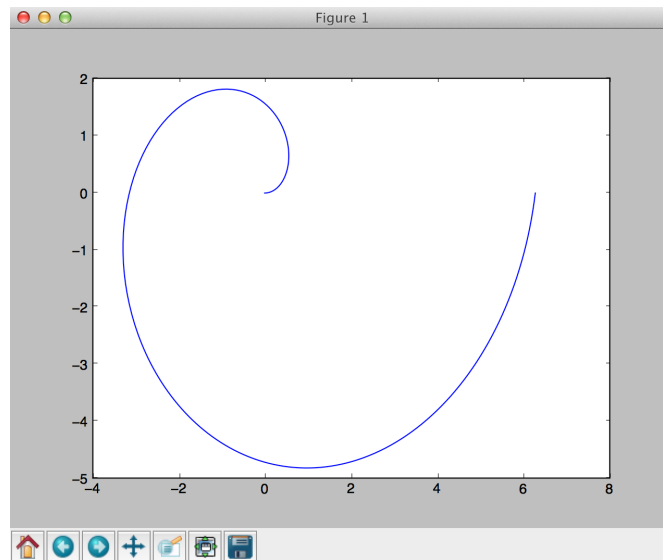
- On améliore le tracé en remplissant quelques options avant de la sauvegarder (au format .png dans le répertoire utilisateur).

```
>>> plt.plot(X, Ycos, 'b', X, Ysin, 'r') # Tracé simultané des 2 courbes
>>> plt.grid(True)                        # Affiche la grille
>>> plt.legend(('cos','sin'), 'upper right', shadow = True) # Légende
>>> plt.xlabel('axe des x')               # Label de l'axe des abscisses
>>> plt.ylabel('axe des y')               # Label de l'axe des ordonnées
>>> plt.title('Fonctions cosinus et sinus') # Titre
>>> plt.savefig('ExempleTrace') # sauvegarde du fichier ExempleTrace.png
>>> plt.show()
```



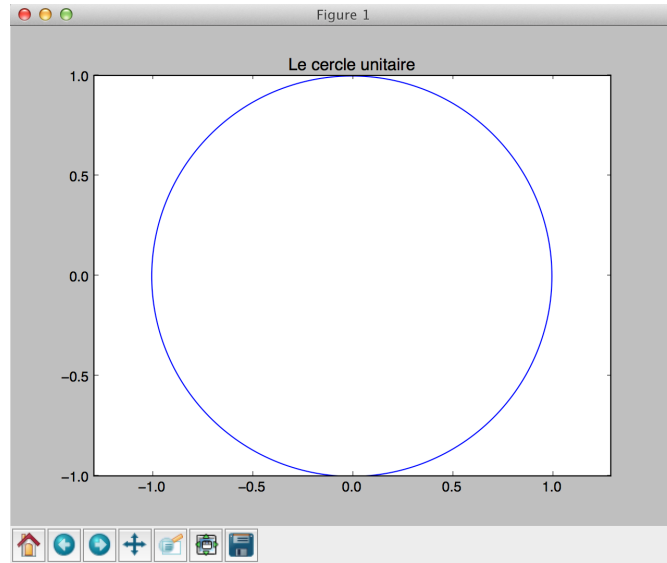
- On peut tout aussi bien tracer des courbes paramétrées.

```
>>> T = np.linspace(0,2*np.pi,256)      # paramètre t
>>> X = [ t * np.cos(t) for t in T ]      # x(t) = t.cos(t)
>>> Y = [ t * np.sin(t) for t in T ]      # y(t) = t.sin(t)
>>> plt.plot(X,Y,'b')                    # Tracé de la courbe paramétrée {(x(t),y(t))}
[<matplotlib.lines.Line2D object at 0x10c044ed0>]
>>> plt.show()
```



- Exemple : Tracé du cercle unitaire.

```
>>> T = np.linspace(0,2*np.pi,256)      # paramètre t
>>> X = np.cos(T)                        # x(t) = cos(t)
>>> Y = np.sin(T)                        # y(t) = sin(t)
>>> plt.plot(X,Y,'b')                    # Tracé de la courbe paramétrée
{(x(t),y(t))}
>>> plt.axis('equal')                    # Pour que le repère soit orthonormé
>>> plt.title('Le cercle unitaire')
>>> plt.show()
```



5.3 scipy

`scipy` est le module à utiliser pour des applications scientifiques élaborées. Notamment pour l'intégration de fonctions (intégrales propres) et la résolution d'équations différentielles. On l'utilisera conjointement à `numpy` car ses fonctions s'appliquent souvent à des tableaux créés sous `numpy`. Tout d'abord importons `numpy` pour bénéficier de la fonction `linspace()` :

```
>>> import numpy as np
```

• Ensuite, pour l'intégration et la résolution d'équations différentielles, nous ne ferons usage que du sous-module `integrate` de `scipy` :

```
>>> from scipy import integrate
```

• `scipy` bénéficie de nombreux autres modules pour d'autres domaines du calcul numérique, algèbre linéaire, statistiques, voir par exemple l'aide en ligne : <http://scipy-lectures.github.io/intro/scipy.html>, ou encore : <http://scipy.org>.

• `scipy.integrate` dispose de différentes méthodes pour l'intégration de fonctions :

1. La méthode `quad(f,a,b)` pour intégrer f sur l'intervalle $[a,b]$:

```
>>> f = lambda x: x**2
>>> integrate.quad(f,0,1)
(0.3333333333333333, 3.700743415417189e-15)
```

Elle retourne un couple constitué de la valeur approchée de l'intégrale (1er élément) et d'une estimation de l'erreur commise. Pour accéder indépendamment à ces 2 valeurs :

```
>>> res = integrate.quad(f,0,1)
>>> res[0]      # Résultat obtenu
0.3333333333333337
>>> res[1]      # Estimation de l'erreur
3.700743415417189e-15
```

2. La méthode `quadrature(f,a,b)` pour intégrer f sur l'intervalle $[a,b]$ (par la méthode de quadrature gaussienne).

```
>>> integrate.quadrature(f,0,1)
(0.33333333333333315, 1.1102230246251565e-16)
```

La deuxième valeur retournée est la différence obtenue entre les résultats de deux dernières itérations (la méthode est récursive et s'arrête dès que cette valeur est inférieure à une borne donnée `tol=1.48e-08` que l'on peut modifier en paramètre optionnel).

3. D'autres méthodes, par exemple pour le calcul d'intégrales doubles ou multiples sont disponibles dans `scipy` :

voir : <http://docs.scipy.org/doc/scipy/reference/integrate.html>.

- Les méthodes suivantes ne s'appliquent pas à une fonction et à un intervalle, mais seulement à un échantillonnage (régulier) de la fonction sur l'intervalle d'intégration :

1. `trapz()` applique la méthode des trapèzes à un échantillonnage :

```
>>> import numpy as np
>>> x = np.linspace(0,1,100)
>>> y = x ** 2
>>> integrate.trapz(y, dx=0.01)
0.33001683501683515
```

Attention à bien passer en argument le pas dx ($= h = \frac{b-a}{n}$) car par défaut $dx = 1$ ce qui aboutirait à un résultat erroné.

2. `simps()` applique la méthode d'intégration de Simpson à un échantillonnage :

```
>>> integrate.simps(y, dx=0.01)
0.33000017005067522
```

3. Si l'échantillonnage est régulièrement espacé et comporte $2^k + 1$ points pour un entier k , alors la méthode `romb()` applique la méthode d'intégration de Romberg à un échantillonnage :

```
>>> x = np.linspace(0,1,257)      # 257 = 2^8 + 1
>>> y=x**2
>>> integrate.romb(y, dx=1/256)
0.3333333333333331
```

Attention à bien spécifier l'option : $dx = 2^k$ (=valeur du pas h), qui vaudrait sinon 1 par défaut et aboutirait à un résultat erroné :

```
>>> integrate.romb(y)      85.333333333333329
```

5.3.1 Intégration d'équations différentielle sous scipy

- La méthode `odeint()` du sous-module `integrate` de `scipy` permet l'intégration d'équations différentielles avec condition initiale :

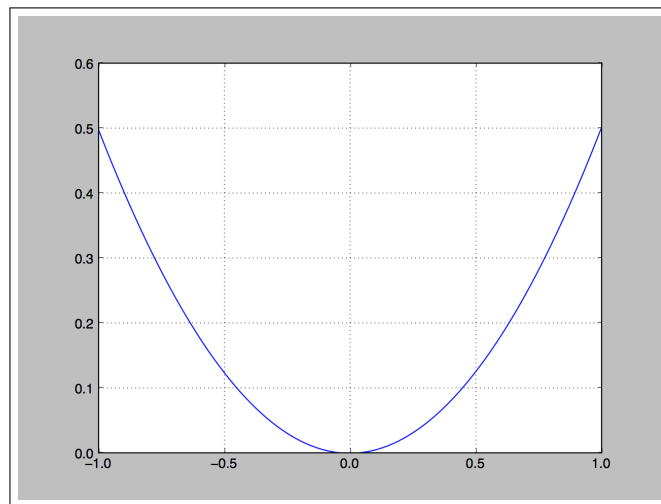
O.D.E. pour 'Ordinary Differential Equation', c.à.d. 'Equation différentielle ordinaire', on dirait plutôt du premier ordre : $y' = \phi(y, t)$ avec condition initiale (t_0, y_0) .

- Utilisation : Prenons l'exemple de l'intégration sur $[-1, 1]$ de l'E.D. :

$$y' = t \quad \text{avec } y(-1) = \frac{1}{2}$$

dont l'unique solution est $y(t) = \frac{t^2}{2}$. (C'est en fait ici une recherche de primitive.)

```
>>> import numpy as np
>>> t = np.linspace(-1,1,100)      # tableau des temps de l'échantillonnage
>>> phi = lambda y,t: t           # fonction du second membre
>>> from scipy import integrate
>>> y0 = 1/2 # condition initiale y[0] = y(-1) = 1/2 à t[0]=-1
>>> y = integrate.odeint(phi, y0, t) # y=échantillon de la fonction sur t
>>> import matplotlib.pyplot as plt # tracé
>>> plt.plot(t,y)
>>> plt.grid()
>>> plt.show()
```



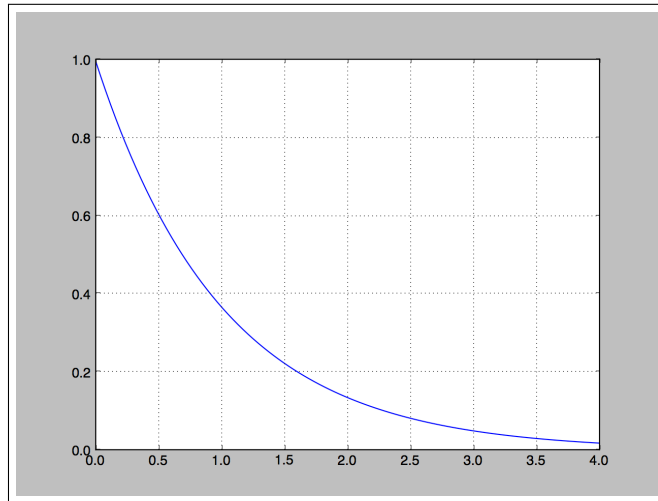
Voici le tracé obtenu. Il correspond bien au résultat attendu : $y(t) = \frac{t^2}{2}$ sur l'intervalle $[-1, 1]$.

- Autre exemple :

$$y' = -y \quad \text{avec } y(0) = 1$$

a pour unique solution $y(t) = e^{-t}$.

```
>>> phi = lambda y,t: -y      # fonction second membre
>>> t = np.linspace(0,4,100)  # 100 échantillonnages entre 0 et 4
>>> y0 = 1                    # valeur initiale au temps t[0] = 0
>>> y = integrate.odeint(phi,y0,t)  # intégration
>>> plt.plot(t,y)             # tracé
>>> plt.grid()
>>> plt.show()
```



5.3.2 Equation différentielles d'ordres supérieurs

- La fonction `integrate.odeint` permet aussi de résoudre des systèmes d'équations différentielles (2 ou plus) :

$$\begin{cases} y_1' = f_1(y_1, t) \\ y_2' = f_2(y_2, t) \end{cases} \quad \text{avec conditions initiales } y_1(x_0) \text{ et } y_2(x_0)$$

Pour cela on passera pour arguments à `odeint` : `integrate.odeint(F,Y0,t)` avec `F` et `Y0` de type array :

`F = np.array([f1,f2])` et `Y0 = np.array([y1(x0),y2(x0)])`.

- C'est cette fonctionnalité que l'on utilise pour résoudre des équations différentielles d'ordre supérieur :

- Exemple :

$$y'' = \cos(t) \quad \text{avec } y(0) = -1 \text{ et } y'(0) = 0$$

a pour solution évidente $y(t) = -\cos(t)$.

L'équation différentielle est équivalente au système (problème de Cauchy) :

$$\begin{cases} \frac{d}{dt}y = y' \\ \frac{d}{dt}y' = \cos(t) \end{cases} \quad \text{avec : } y(0) = -1 \text{ et } y'(0) = 0$$

• Le système :

$$\begin{cases} \frac{d}{dt}y = y' \\ \frac{d}{dt}y' = \cos(t) \end{cases} \quad \text{avec : } y(0) = -1 \text{ et } y'(0) = 0$$

s'écrit comme un problème de Cauchy vectoriel :

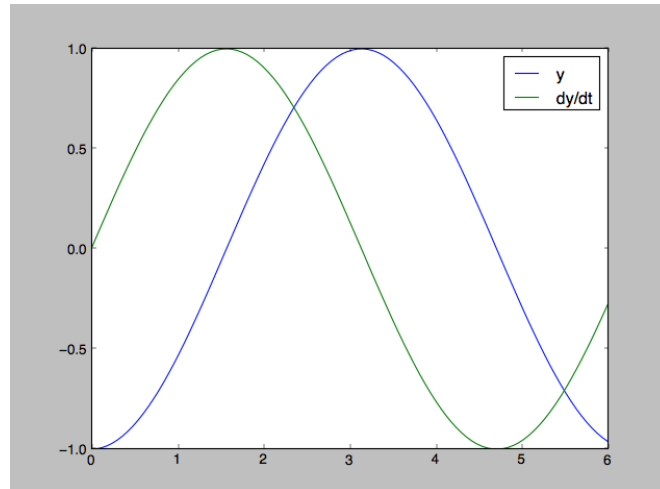
$$\frac{d}{dt} \begin{pmatrix} y \\ y' \end{pmatrix} = \begin{pmatrix} y' \\ \cos(t) \end{pmatrix} \quad \text{et} \quad \begin{pmatrix} y(0) \\ y'(0) \end{pmatrix} = \begin{pmatrix} -1 \\ 0 \end{pmatrix}$$

En posant $Y = \begin{pmatrix} y \\ y' \end{pmatrix} = \begin{pmatrix} Y[0] \\ Y[1] \end{pmatrix}$:

```
import numpy as np
def F(Y,t):          # array des fonctions aux seconds membres
    return np.array([Y[1],np.cos(t)])
from scipy import integrate
t = np.linspace(0,6,100)    # Subdivision régulière de [0,6]
y0 = np.array([-1,0])      # array des conditions initiales
y = integrate.odeint(F,y0,t)    # appel de odeint
```

Tracé du graphe :

```
import matplotlib.pyplot as plt
plt.plot(t,y)
plt.legend(('y','dy/dt'))
plt.show()
```



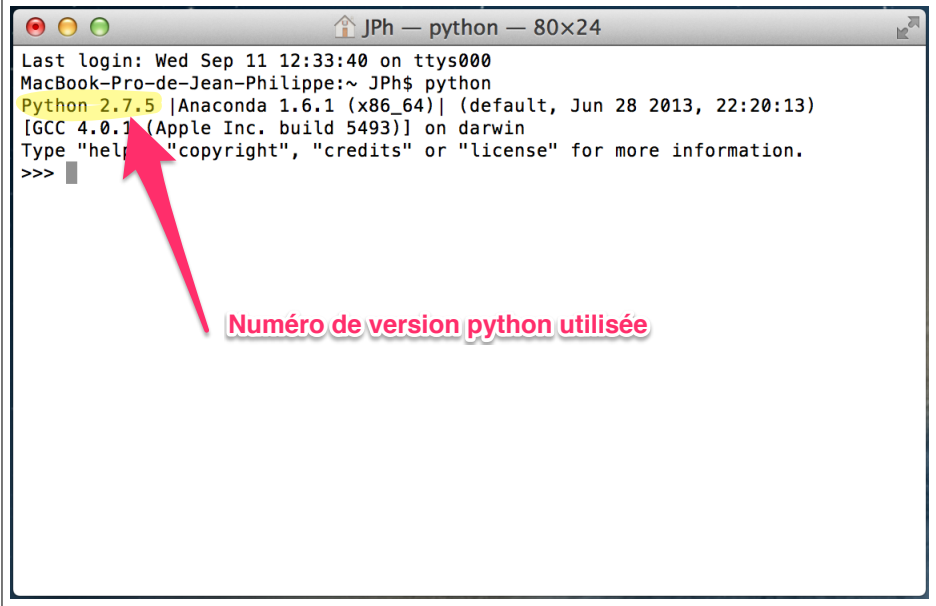
On obtient 2 courbes, celle de $y(t) = -\cos(t)$ et celle de $y'(t) = \sin(t)$.

Annexe A

python 2 versus python 3

A.1 python versions 2 et 3

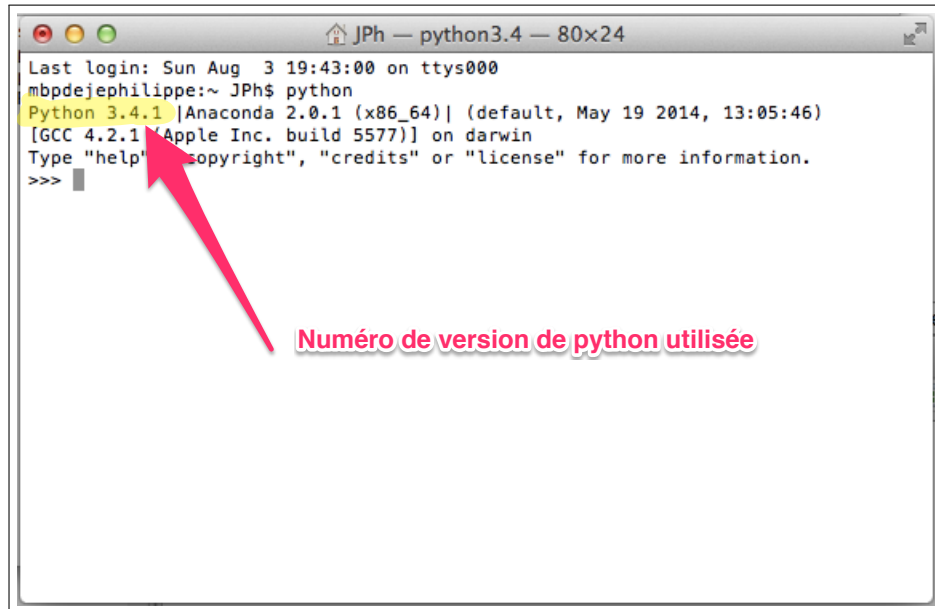
- En 2008, sort la version 3 de **python** qui perd sa compatibilité avec la version 2 précédente du langage, afin d'en corriger les principaux défauts.
- On peut encore rencontrer les versions 2 (ver. 2.7) et 3 (ver. 3.4) de **python**.
- Si la version 2 est encore largement utilisée c'est d'abord car on perd la compatibilité entre les versions 2 et 3, et surtout car tous les modules de **python** ne sont pas encore, à ce jour, compatibles avec la version 3.
- Si l'on débute en **python**, et si l'on n'a pas besoin des quelques rares modules qui ne sont pas encore compatibles, il vaut mieux utiliser la dernière version, la version 3.
- On peut vérifier quelle version de **python** est utilisée lors de son lancement :



A terminal window titled 'JPh — python — 80x24' showing the output of the 'python' command. The output includes the last login time, the command executed, the Python version (2.7.5), the Anaconda version (1.6.1), the default environment, the GCC version (4.0.1), and the operating system (darwin). A red arrow points to the '2.7.5' in the output, with a label 'Numéro de version python utilisée'.

```
Last login: Wed Sep 11 12:33:40 on ttys000
MacBook-Pro-de-Jean-Philippe:~ JPh$ python
Python 2.7.5 |Anaconda 1.6.1 (x86_64)| (default, Jun 28 2013, 22:20:13)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- Ci-dessus il s'agit de la version 2.7.5, ci-dessous de la version 3.4.1 :

A terminal window titled 'JPh — python3.4 — 80x24'. The output shows the last login time, the command 'python' being executed, and the resulting Python version information: 'Python 3.4.1 |Anaconda 2.0.1 (x86_64)| (default, May 19 2014, 13:05:46) [GCC 4.2.1 (Apple Inc. build 5577)] on darwin'. A red arrow points from the text 'Numéro de version de python utilisée' to the '3.4.1' part of the output.

```
Last login: Sun Aug 3 19:43:00 on ttys000
mbpdejephilippe:~ JPh$ python
Python 3.4.1 |Anaconda 2.0.1 (x86_64)| (default, May 19 2014, 13:05:46)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "help" for copyright, "credits" or "license" for more information.
>>>
```

- On peut aussi obtenir ces mêmes informations, dont la version utilisée, grâce aux instructions :

```
>>> import sys
>>> sys.version
'3.4.1 |Anaconda 2.0.1 (x86_64)| (default, May 19 2014, 13:05:46) \ n[GCC
4.2.1 (Apple Inc. build 5577)]'
```

- Ici il s'agit encore de la version python 3.4.1.

A.2 Principaux changements de la version 3

A.2.1 Changements de la fonction print()

- La commande `print` n'existe plus, au seul profit de la fonction `print()` :

En python 2 les deux syntaxes suivantes sont acceptées :

```
>>> print "bonjour", "le monde"
bonjour le monde
>>> print("bonjour", "le monde")
bonjour le monde
```

En python 3 seule la deuxième syntaxe est acceptée :

```
>>> print("bonjour", "le monde")
bonjour le monde
```

`print` est obligatoirement suivie de parenthèses.

A.2.2 Changements de la fonction `input()`

- En python 2 `input()` convertit l'entrée saisie dans le type adéquat :

```
>>> a = input('? ' )
? 3.1415
>>> type(a)
<type 'float'>
>>> print(a+1)
4.1415
```

Une chaîne de caractère doit être saisie entre `'.'` ou `"."` :

```
>>> a = input('? ' )
? 'bonjour'
>>> type(a)
<type 'str'>
>>> print(a)
bonjour
```

- Pour saisir une chaîne de caractère sans `'.'` ou `"."` il faut plutôt utiliser la fonction `raw_input()` :

```
>>> a = raw_input('? ')
? bonjour
>>> print(a)
bonjour
```

- En python 3 `input()` retourne toujours une chaîne de caractère qui ne nécessite plus d'être saisie entre `'.'` ou `"."`.

```
>>> a = input('? ' )
? 3.1415
>>> type(a)
<class 'str'>
>>> print(float(a)+1)
4.1415
```

En contrepartie il faut une conversion à l'aide de `int()` ou `float()` si l'on souhaite un nombre.

En python 3 `input()` se comporte comme la fonction `raw_input()` de python 2. **En python 3 la fonction `raw_input()` n'existe plus.**

A.2.3 Changement de l'opération de division `/`

- En python 2 la division `/` entre nombre entiers retourne un nombre entier :

```
>>> 1 / 2
0
```

- Pour obtenir un flottant, le résultat correct, il faut qu'au moins un des deux opérandes soit de type float :

```
>>> 1 / float(2)
0.5
>>> 1.0 / 2
0.5
>>> 1. / 2
0.5
```

- En python 3 ce n'est plus le cas :

```
>>> 1 / 2
0.5
```

- Pour obtenir le quotient entier de deux entiers en python 3 il faut utiliser l'opérateur `//` :

```
>>> 1 // 2
0
```

- Pour imposer qu'en python 2 la division se comporte comme en python 3 taper l'instruction d'importation :

```
>>> from __future__ import division
```

A.2.4 Changements de la fonction `range()`

- En python 2 la fonction `range()` retourne une liste :

```
>>> range(0,10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- Ce n'est plus le cas en python 3 : `range()` retourne un itérateur :

```
>>> range(0,10)
range(0,10)
```

c'est à dire la méthode pour calculer les différents éléments, afin d'alléger l'usage de la mémoire.

Pour obtenir une liste en python 3 utiliser la fonction de conversion `list()` :

```
>>> list(range(0,10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- La fonction `range()` de python 3 a le même résultat que la fonction `xrange()` de python 2. **La fonction `xrange()` n'existe plus en python 3.**