1.1 Introduction

1.1.1 Le langage python

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

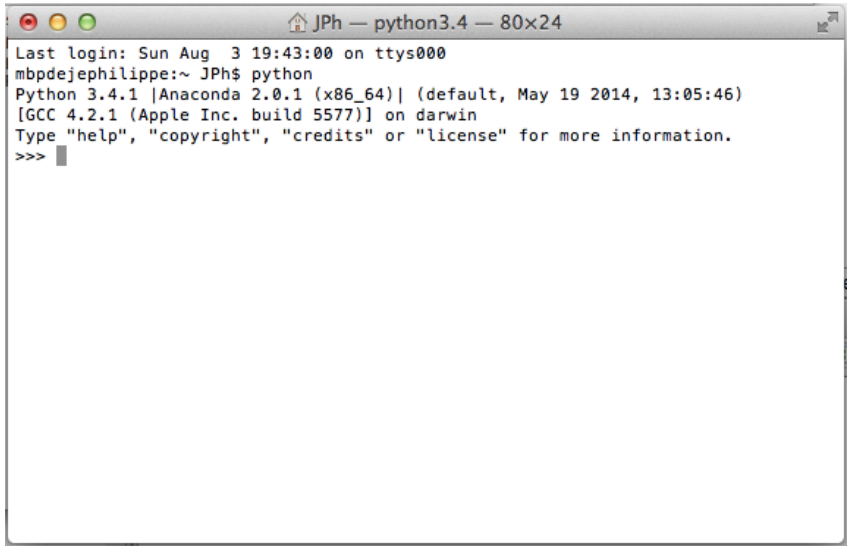
Il présente les avantages suivants :

- 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

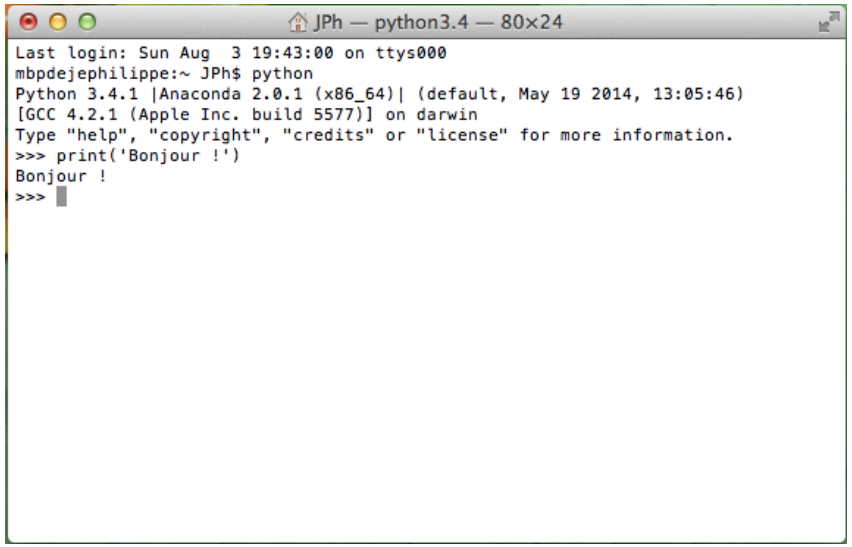
Python est un langage interprété qui peut être utilisé en mode console. Lancer une fenêtre de console (terminal, console ou fenêtre de commande selon le Système d'exploitation) et simplement taper à l'invite 'python' suivi de la touche entrée.



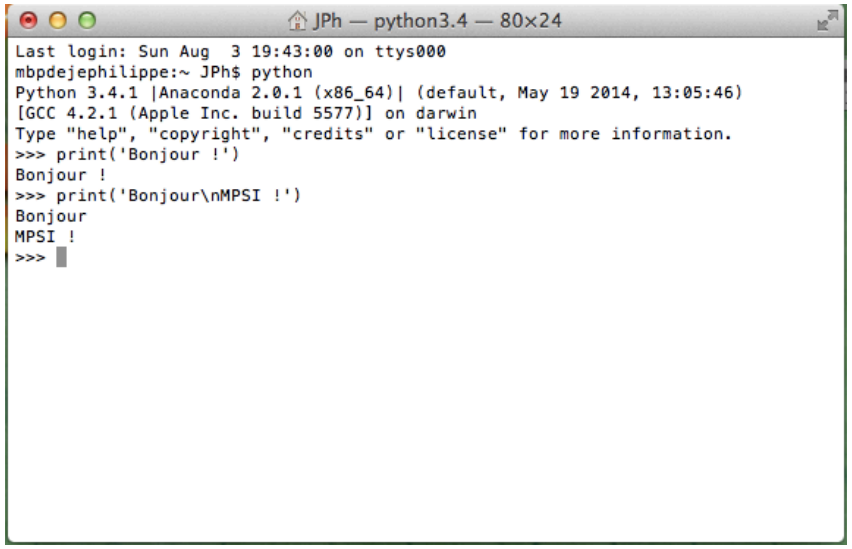
Un prompt, symbolisé ici par : >>>, apparaît. On peut y saisir tout type de commandes python.

Une console au lancement de python.

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



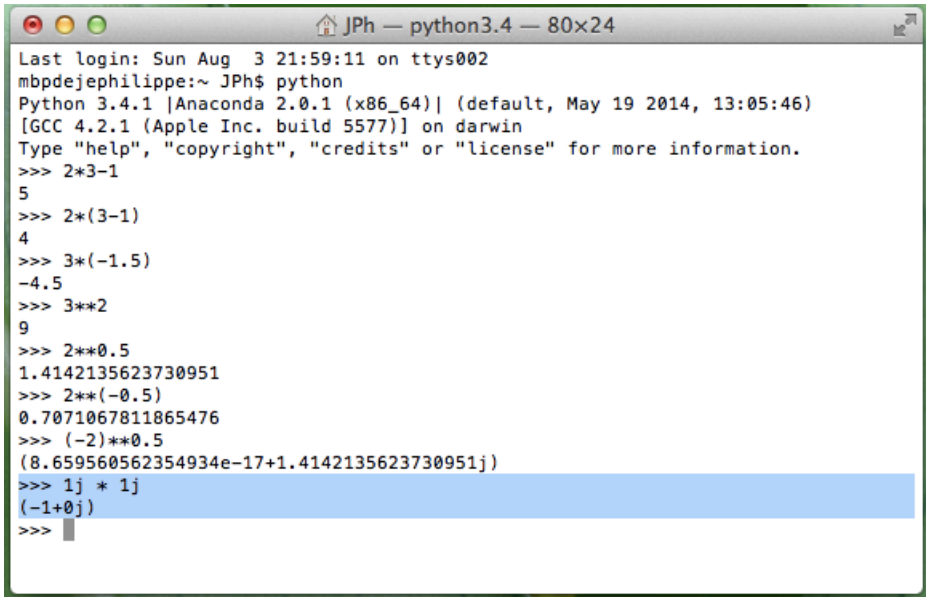
La commande (ou fonction) 'print(.)' écrit à l'écran une chaîne de caractère, c'est à dire une suite finie de caractères entre apostrophes '...' ou double-quotes "...".



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

1.2.2 Calculatrice

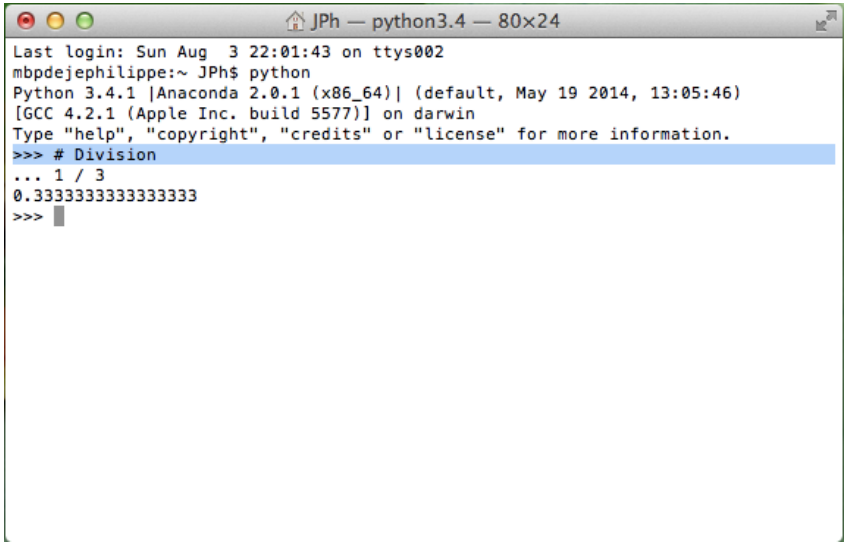
Python peut se comporter comme une calculatrice.



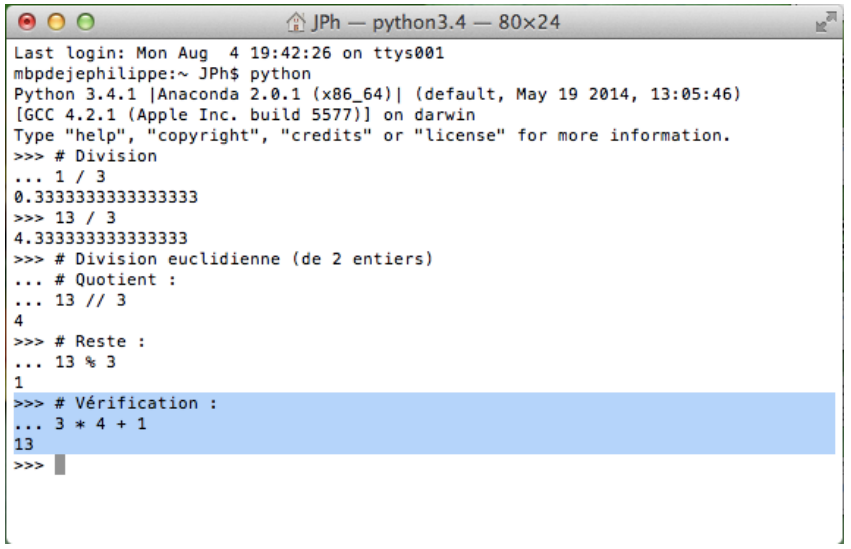
- Il respecte l'ordre usuel des opérations.
- Et comprend les nombres à virgule flottante 'de type float'.
- La mise en puissance s'obtient grâce à l'opérateur `**`.
- L'exposant peut être 'réel'. Ainsi l'opérateur `**0.5` extrait la racine carrée.
- Lorsque y n'est pas entier, le réel x^y n'est défini que pour $x > 0$: `python` (version 3) retourne un nombre complexe, valeur approchée ici de $i\sqrt{2}$.
- le complexe $x + i.y$ s'écrit `x+yj` (avec `x,y` des flottants).

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

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

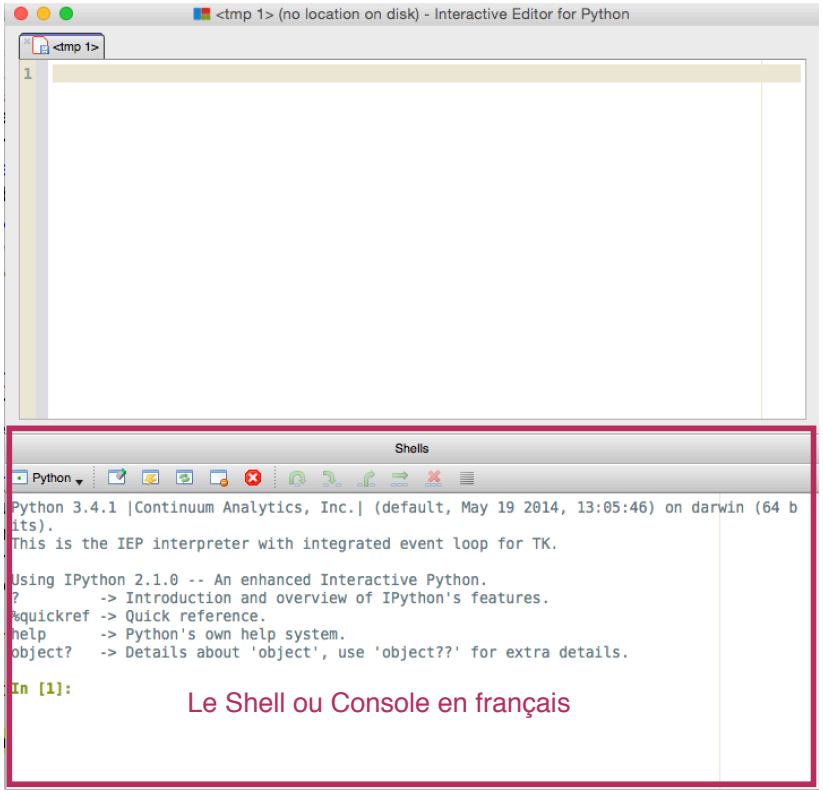


- `/` est l'opérateur de division.
- `//` est l'opérateur de quotient de 2 entiers n et $m \neq 0$ dans la division euclidienne de n par m ; il retourne le quotient entier $n//m = \lfloor \frac{n}{m} \rfloor$.
- L'opérateur `%` retourne le reste de la division euclidienne : $n = (n//m) \times m + (n\%m)$.



1.2.3 Dans la console de Pyzo

Une console, dénommée "shell" est présente dans l'EDI.
Sous Pyzo : il est ici symbolisé dans le rectangle rouge :



1.2.4 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 biliothèque
>>> pi
3.141592653589793
>>> cos(pi)
-1.0
>>> acos(-1)
3.141592653589793
```

```
>>> 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 abolue
3.0
>>> floor(pi)                   # partie entière
3.0
>>> floor(-pi)
-4.0
```

1.2.5 Le module `fractions`

Le module `fractions` permet le calcul sur les fractions :

Cet exemple se passe de commentaires :

```
>>> # Le module fractions
... 1/3 + 2/5
0.7333333333333334
>>> from fractions import Fraction
>>> Fraction(1,3) + Fraction(2,5)
Fraction(11, 15)
```

Ici l'instruction '`from fractions import Fraction`' n'importe que la fonction `Fraction` de la bibliothèque `fractions`. On peut aussi importer toute la bibliothèque à l'aide de : '`from fractions import *`'.
('`*`' se lit 'all' = 'tout').

1.2.6 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 maProcedure(x):
...     print("L'image par maFonction de", x, "est", maFonction(x))
...
>>> maProcedure(1)
L'image par maFonction de 1 est 0
>>> maProcedure(0)
L'image par maFonction de 0 est 1
>>> maProcedure(10)
L'image par maFonction de 10 est 81
```

Remarquer qu'une fonction peut être appelée dans la définition d'une autre 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 maProcedure(x):
...     print("L'image par maFonction de", x, "est", maFonction(x))
...
>>> maProcedure(1)
L'image par maFonction de 1 est 0
>>> maProcedure(0)
L'image par maFonction de 0 est 1
>>> maProcedure(10)
L'image par maFonction de 10 est 81
```


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

Syntaxe pour la définition d'une fonction :

```
def NomdelaFonction(Paramètres):
.....   Instruction1
.....   Instruction2
.....   :
.....   Dernière instruction
```

meme espace — bloc d'instructions

L'instruction '`def`' permet de définir une fonction. Elle est suivie du nom de la fonction obligatoirement suivi de parenthèses pouvant contenir des noms de variables, séparées si nécessaire de virgules (ce sont ses paramètres). La parenthèse fermante est suivie de deux points ':'. Suit un bloc d'instructions. Elles doivent toutes être décalées du même nombre d'espaces (en général 3 ou 4). Appuyer sur entrée au prompt pour achever la définition.

Le résultat de la fonction, si il y a, est retourné grâce à l'instruction '`return`'. Sinon on peut parler de procédure.

1.2.7 Variables

- La notion de variable est essentielle en programmation.
- Elle permet de stocker en mémoire des valeurs, et de les utiliser et modifier à volonté au sein d'un programme.
- La valeur d'une variable évolue au cours de l'exécution d'un programme, en fonction du déroulement du programme et selon ses instructions.

```
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"
>>>
```

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 symbôle `=`.

```
>>> # 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 symbôle `=` est évaluée, avant d'être affecté à la variable dont le nom figure à gauche du symbôle `=`.

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 :

`variable †= expression`

est équivalente à l'instruction :

`variable = variable † expression`

où † 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 chaine de caractère alphanumérique, c.à.d. composée de lettres et de chiffres (et du symbôle '_'), qui ne doit pas débuter par un chiffre. Eviter de le débuter par une lettre majuscule, que l'on réservera aux fonctions. Son nom doit être clair pour faciliter la relecture du programme.

– Un **contenu**, c'est sa valeur.

Elle est stockée dans la mémoire sous forme d'un nombre en écriture binaire, pour l'interpréter correctement, chaque valeur est munie d'un **type** : entier (relatif), flottant (réel...), complexe, chaîne de caractère, etc...

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 de la valeur 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* : Le type de sa valeur peut être modifié.

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

C'est cependant à déconseiller !

1.2.8 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 :

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

```
>>> # 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.

1.2.9 Échange du contenu de deux variables

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
```

Sans variable temporaire :

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

il faut tout de même 3 affectations, et le code n'est pas facilement lisible... Etat de la mémoire durant l'exécution :

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

1. `a, b = 1, 2`

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

2. `a = a+b`

`a`	$\alpha + \beta$ (=3)
`b`	β (=2)

3. `b = a-b`

`a`	$\alpha + \beta$ (=3)
`b`	$(\alpha + \beta) - \beta = \alpha$ (=1)

4. `a = a-b`

`a`	β (=2)
`b`	α (=1)

La deuxième ligne a permis d'échanger les valeurs contenues dans les variables `a` et `b`.