CHAPITRE 2

Premiers pas

Dans ce chapitre, on ne se préoccupe pas d'être exhaustif, ni rigoureux, mais plutôt de faire un tour d'horizon du langage et de tester quelques instructions élémentaires afin de s'imprégner de l'esprit et des constructions ***Python***. Dans les chapitres suivants, nous reviendrons plus en détail sur tous les éléments présentés ici.

Ouvrez une session ***Python*** puis testez, jouez avec les instructions suivantes.

2.1. Pour vérifier votre installation

Vous pouvez tester ces instructions :

```
>>> import sys
>>> sys.argv
['']
```

On importe le module `sys`, et on consulte son attribut `sys.argv` qui contient les arguments de la ligne de commande quand ***Python*** a été lancé. Ici, la liste d'arguments est vide.

Les lignes de continuation sont nécessaires lorsqu'on saisit une construction sur plusieurs lignes. Comme, par exemple, dans cette instruction ***if*** :

```
>>> a = 1
>>> if a  :
...   print ("1 est vrai")
...
```

On peut customiser ses prompts, par exemple les mettre en couleur rouge[1] :

```
>>> sys.ps1 = 'moi##> '
moi##> sys.ps1 = '\033[95m>>> \033[0m'
>>> sys.ps2 = '$$$ '
>>> help(
$$$ )
```

Pour interrompre une instruction trop longue à s'exécuter taper `Ctrl+c`

```
>>> while 1  :
...   print(3.14)
<Ctrl+c>
Traceback (most recent call last) :
  [...]
KeyboardInterrupt
```

[1] Dans *Ipython*, ainsi que dans *idle* il semble que `sys.ps1` et `sys.ps2` n'existent plus

2.2. Aide en ligne, *dir() et help()*

`help()` est une fonction importante qui, comme son nom l'indique, donne accès à l'aide en ligne :

```
>>> help
Type help() for interactive help, or help(object) for help
   about object.
```

`help(obj)` permet de consulter l'aide sur l'objet *obj*.

Pour consulter l'aide sur le module *sys* :

```
>>> import sys
>>> help(sys)
Help on built-in module sys :
NAME
sys
FILE
(built-in)
          [etc ...]
warnoptions = []
```

On voit que l'aide est consistante, et qu'elle décrit en particulier les fonctions du module *sys* que l'on a importé auparavant.

La fonction `dir()` sans paramètre, donne la liste des noms accessibles dans la portée actuelle. Pour savoir ce que représentent ces noms, tapez simplement le nom :

```
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'help',
   'os', 'sys']
>>> sys
<module 'sys' (built-in)>
```

On y voit en particulier le module *sys* importé, et que c'est un module *builtin*.

Avec un *objet* en paramètre, `dir(objet)` retourne une liste de tous les attributs (méthodes et données) valides de cet objet. Le nombre entier 1 est un objet ***Python***. En ***Python*** tout est objet au sens programmation orientée objet. À ce titre, tout objet possède des attributs. Les attributs d'une variable peuvent être consultés :

```
>>> dir(1)
['__abs__', etc..., 'numerator', 'real']
>>> a = 1
>>> dir(a)
['__abs__', etc..., 'numerator', 'real']
>>> a.numerator
1
```

Pour savoir ce que fait réellement la fonction `dir()`, on peut demander de l'aide :

```
>>> help(dir)
Help on built-in function dir in module builtins :
dir(...)
   dir([object]) -> list of strings
   [...]
```

2.3. Commentaires

En ***Python***, est commentaire (ignoré par l'interpréteur) tout ce qui va du caractère # à la fin de la ligne, à condition que le # ne fasse pas partie d'une chaîne de caractères.

```
>>> # voici le premier commentaire
>>> truc = 1 # et voici le deuxième commentaire
>>> STRING = "# Ceci nest pas un commentaire \
          mais une chaîne de caractères \
          sur trois lignes "
```

2.4. Indentation

Dans les langages comme ***C++*** , *Pascal* ou *Java*, le corps d'une boucle est déterminé une paire d'accolades {} ou bien `begin...end`

En ***Python***, c'est l'indentation qui est interprétée pour définir le corps de la boucle.

Si l'on démarre le corps de la boucle avec une indentation de 4 (quatre) espaces (ce qui est la valeur recommandée), alors toutes les autres instructions du corps de la même boucle devront être indentées de 4 (quatre) espaces. C'est une erreur de syntaxe d'indenter deux instructions d'une valeur différente dans un même corps de boucle.

Par exemple ce programme confus contient au moins trois erreurs de syntaxe :

```
>>> L ='€€€€€€$$$$$$$$$$'
>>> n= 0
>>>     for c in L :
... print (c)
...  if c == '€' :
... print 'encore un euro'
...    n= n+1
...      print ('comptage terminé')
```

il est donc illisible pour l'interpréteur ***Python et*** illisible pour l'œil humain.

Alors que le programme qui suit est beaucoup plus clair et exempt d'erreur de syntaxe.

```
>>> L = '€€€€€€$$$$$$$$$$'
>>> n = 0
>>> for c in L :
...  if c == '€' :
...     print ('encore␣un␣euro')
...     n += 1
... print('comptage␣terminé')
```

Python nous *oblige* à présenter les programmes de manière lisible pour lui *et* pour l'œil humain.

Attention de ne pas mélanger les tabulations et les espaces car cela donne lieu à des erreurs de compilation qui peuvent paraître incompréhensibles.

2.5. Variables, types

On peut déclarer des variables dont le type est déterminé à l'affectation. La fonction intégrée `type()` permet de connaître le type (ou la classe) d'une variable :

```
>>> a = 2
>>> type(a)
<class 'int'>
>>> x = 3.14
>>> type(x)
<class 'float'>
>>> import math
>>> x - math.pi
-0.0015926535897929917
```

a est de type entier, *x* de type réel.

Le module *math* contient la constante *math.pi*.

Une variable ***Python*** est constituée d'un nom, d'un type et d'une valeur.

L'opérateur d'affectation est `'='`. Une instruction comme :

```
a = 2
l = [1,2,3]
```

affecte la valeur entière 2 à la variable *a*. Ce qui signifie que *a* est un nom donné à l'entier 2, et a pour type **int**. La variable *a* est donc `('a',int,2)`

De même *l* est un nom donné à la liste `[1, 2, 3]`, de type **list**.

On peut faire une affectation multiple de deux manières différentes :

```
>>> x = y = z = 0 # Mettre a zero x, y et z
>>> x, y, z = 0, 0, 0 #idem
>>> x
0
```

Il est possible de déclarer des variables plus élaborées comme des tuples, des listes ou des dictionnaires :

```
>>> T = (1,2,3)
>>> type(T)
<class 'tuple'>
>>> L = [1,2,3]
>>> type(L)
<class 'list'>
```

```
>>> D = {}
>>> type(D)
<class 'dict'>
```

On examinera plus loin ces objets ***Python***.

Le simple fait d'entrer au clavier le nom d'une variable affiche son contenu. Une alternative consiste à utiliser la fonction ***print***().[2]

```
>>> a,b = 1,3.14
>>> a,b
(1, 3.14)
>>> print ("a et b valent" , a,'et', b)
a et b valent 1 et 3.14
```

Outre les entiers (**int**) et les réels (**float**), les complexes (**complex**) font également partie du langage de base de ***Python*** :

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
>>> a = 1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
>>> abs(a)
1.5811388300841898
>>>
```

Quelques variables sont prédéfinies :

```
>>> import math, sys
>>> math.pi
3.141592653589793
>>> sys.float_info[0]
1.7976931348623157e+308
>>> sys.float_info.min
2.2250738585072014e-308
```

La variable _ (« *underscore* ») contient la dernière valeur calculée :

[2] En ***Python*** 2 ***print*** est une *instruction* et utilise la syntaxe ***print*** *x* sans parenthèse.

```
>>> tva = 12.5 / 100
>>> prix = 100.50
>>> prix * tva
12.5625
>>> prix + _
113.0625
```

2.6. Opérations

Les opérateurs suivants sont définis pour les types **`int`**, **`float`**, et **`complex`** :

x+y : addition
x-y : soustraction
x*y : multiplication
x/y : division
x//y : division entière (**`float`** et **`int`** seulement)
x%y : reste de la division entière (**`float`** et **`int`** seulement)
xy :** élévation à la puissance x^y

Les opérations arithmétiques se comportent de manière conventionnelle, même la division.[3]

```
>>> 2*5.1
10.2
>>> 0+3.14
3.14
>>> 3**2
9
>>> pow(3,2)
9
>>> 7/3
2.3333333333333335
>>> 7.0/3.0
2.3333333333333335
```

La fonction **`divmod`**`(a,b)` retourne un couple de valeurs (quotient, reste), résultat de la *division entière* de *a* par *b*. Il s'agit de l'unique couple (q, r), de $\mathbb{N} \times \mathbb{R}$ vérifiant $a = bq + r$ avec $0 \leqslant r < b$. Les arguments *a* et *b* peuvent être réels ou entiers.

Les opérateurs `//` et `%` donnent séparément le quotient et le reste de la division entière.

```
>>> 7//3
2
>>> 7%3
1
```

[3] Attention,

- en ***Python*** 2, la division `/` appliquée à des entiers retourne le résultat de la division *entière ;*
- en ***Python*** 3, l'opérateur `/` retourne toujours le résultat de la division réelle. L'opérateur `//` retourne le résultat de la division entière.

L'opérateur de comparaison == permet de tester l'égalité de deux objets, ici des couples.

```
>>> divmod(7,3) == (7//3, 7%3)
True
```

La division entière peut s'appliquer aux nombres réels (**float**) :

```
>>> 7.0//3.2
2.0
>>> 7.0%3.2
0.5999999999999996
```

2.7. Chaînes de caractères

Les *chaînes de caractères* sont le type **str** de ***Python***. Une chaîne de caractères est une suite finie de caractères placés

- entre apostrophes (ou *simples quotes*) ' ou
- entre triples apostrophes ''' ou
- entre guillemets (ou *doubles quotes*) " ou
- entre triples guillemets """.

Les chaînes délimitées par des triples apostrophes ou des triples guillemets sont appelées *docstring* ou *chaînes de documentation*. Certaines fonctions (en particulier la fonction **help**()), certains modules de documentation (le module *doctest*) utilisent ces docstrings pour extraire la documentation des fonctions, classes, modules, etc..

```
>>> 'spam eggs'
'spam eggs'
>>> 'Une chaîne qui \
a une suite'
'Une chaîne qui a une suite'
>>> "une chaîne entre guillemets"
'une chaîne entre guillemets'
>>> salut = """Ceci est une chaîne plutôt longue contenant
plusieurs
lignes de texte
  Notez que les blancs au début de la ligne et les sauts de
     ligne
sont significatifs."""
>>> print (salut)
Ceci est une chaîne plutôt longue contenant
plusieurs
lignes de texte exactement
  Notez que les blancs au début de la ligne et les sauts de
     ligne
sont significatifs.
```

Placé en fin de ligne, le caractère `'\'` (*antislash*) sert de « carte suite » pour indiquer à l'interpréteur que la ligne courante et la suivante doivent être considérées comme une seule ligne logique. C'est le cas pour une ligne trop longue par exemple.

Pour « échapper » un caractère ayant une signification particulière, (par exemple une apostrophe `'` fermant une chaîne de caractères) on le fait précéder du caractère `'\'` (antislash ou *backslash*). Il perd sa signification particulière, et devient un caractère standard.

```
>>> 'n\'est-ce pas'
"n'est-ce pas"
>>> "n'est-ce pas"
"n'est-ce pas"
>>> '"Oui," dit-il.'
'"Oui," dit-il.'
>>> "\"Oui,\" dit-il."
'"Oui," dit-il.'
>>> '"N\'est-ce pas," répondit-elle.'
'"N\'est-ce pas," répondit-elle.'
```

Les chaînes peuvent être concaténées (accolées) avec l'opérateur `'+'`, et répétées avec `'*'` :

```
>>> word = 'Help' + 'A'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
```

Il n'existe pas de type *caractère* en ***Python***, un caractère est simplement une chaîne de longueur un.

```
>>> type('a')
<class 'str'>
>>> type('aaa')
<class 'str'>
```

La fonction intégrée `len()`, appliquée à une chaîne de caractères retourne le nombre de caractères de la chaîne :

```
>>> len('anticonstitutionnellement')
25
```

Les éléments d'une chaîne de caractère *ch* sont indexés (numérotés) de `0` à `len(ch)-1`. On peut accéder à chaque caractère par l'*opérateur crochet* `[]` :

```
>>> ch = 'anticonstitutionnellement'
>>> ch[0]
'a'
>>> ch[len(ch)] #ouuuups
Traceback (most recent call last) :
  [...]
    ch[len(ch)] #ouuuups
IndexError : string index out of range
```

Les éléments d'une chaîne de caractères sont *non modifiables*.