Chapitre 1

Les bases de python

1.1 Hello World

Voici le plus simple des programmes, le « Hello World », Nous allons profiter de cet exemple pour mettre en valeur la concision de l'écriture de code en Python. Nous souhaitons donc simplement afficher dans une console le message

« Hello World ! ». En C, nous aurons besoin de six lignes :

Hello word en C

```c
#include <stdio.h>

int main(void)
{
printf("Hello world!\n ");
return 0;
}
```

Il faudra bien sûr ensuite compiler et exécuter le programme :

```
meilland@debian:~$ gcc -Wall hello.c
meilland@debian:~$ a.out
Hello llorld  !
```

Cela représente beaucoup de travail pour un petit résultat !

Et encore, on peut faire encore plus long avec du Java :

Hello word en java

```java
public class Hello
{
        public static void main(string[] args)
        {
                System.out.println("Hello World!\n");
        }
}
```

ici nous devons créer une classe et donc utiliser la programmation orientée objet pour afficher un simple message ! Et nous devons bien sûr passer par une étape de pré-compilation :

```
login@serveur~$ javac Hellojava
login@serveur~$ java Hello
Hello World !
```

C'est toujours très long pour afficher un simple message. Voyons maintenant le code Python permettant d'obtenir le même résultat. À ce stade, nous utiliserons l'éditeur interactif en lançant

python dans un terminal, puis en tapant nos commandes Python. Pour commencer, en Python 3.2 nous entrerons le code :

```
print("Hello World !")
```

Oui, c'est tout et ça paraît logique puisque tout ce que je demande c'est l'affichage de ce message ! Le résultat de l'interprétation de cette commande dans l'interpréteur interactif est montré en figure 7.

" « Hello World » en Python dans un interpréteur interactif"

```
>>> print("Hello World !")
Hello World !
```

Remarquez au passage l'absence de point-virgule en fin de ligne, contrairement au C, à Java et à bon nombre de langages.

Maintenant que vous avez vu votre premier programme - ou script - en Python, certes court, mais programme quand même, je vous propose d'explorer les différents types de données manipulables en Python.

1.2 Les différents types de données

Nous avons vu que Python était un langage fortement typé utilisant le *duck typing* pour déterminer le type des données. Pour tester les types présentes dans cette partie, vous pourrez utiliser l'interpréteur interactif. C'est d'ailleurs sous cette forme que je présenterai les différents exemples en faisant précéder les commandes des caractères » » » correspondant au prompt de l'interpréteur.

Pour pouvoir travailler sur les types, nous aurons besoin de faire appel à une fonction : la fonction *type*(). Le rôle de cette fonction est d'afficher le type de l'élément qui lui est passé en paramètre.

1.2.1 Les données numériques

Parmi les données manipulables, on va bien entendu pouvoir utiliser des nombres. Ceux-ci sont classes en quatre types numériques :

- Entier : permet de représenter les entiers sur 32 bits (de −2147483648 à 2147483647).

" « Hello World » en Python dans un interpréteur interactif"

```
>>> type(2)
<type 'int'>
>>> type(2147483647)
<type 'int'>
```

Des préfixes particuliers permettent d'utiliser la notation octale (en base huit) ou hexadécimale (en base seize). Pour la notation octale, il faudra préfixer les entiers par le chiffre 0 en Python 2.7 et par 0*o* (le chiffre zéro suivi de la lettre « o » en minuscule) en Python 3.2. Par exemple, 12 en notation octale vaut 8+2=10 :

" « Hello World » en Python dans un interpréteur interactif"

```
>>> 0o12
10
```

Pour la notation hexadécimale, le préfixe sera le même que l'on soit sous Python 2.7 ou Python 3.2, ce sera ***0x*** (le chiffre zéro suivi de la lettre « x » en minuscule). Ainsi, 12 en notation hexadécimale vaut 16+2=l8 :

" « Hello World » en Python dans un interpréteur interactif"

```
>>> 0x12
18
```

- Entier : On peut forcer un nombre ou texte en entier avec la fonction *int*()

  " « Hello World » en Python dans un interpréteur interactif"

  ```
  >>> type(int("16"))
  <class 'int'>
  ```

- Flottant : pour les nombres flottants. Le symbole utilisé pour déterminer la partie décimale est le point. On peut également utiliser la lettre ***E*** ou ***e*** pour signaler un exposant en base 10 (101, 102, etc.).

" « Hello World » en Python dans un interpréteur interactif"

```
>>> type(3.14)
<class 'float'>
>>> 2e1
20.0
>>> type(2e1)
<class 'float'>
```

- Complexe : pour les nombres complexes qui sont représentés par leur partie réelle et leur partie imaginaire.
  En Python, la partie imaginaire sera logiquement suivie de la lettre ***J*** ou ***j***. En informatique, on utilise souvent la variable i comme variable de boucle (dont nous parlerons par la suite), Pour éviter toute confusion, la partie imaginaire des complexes se note donc j. Voici comment écrire le complexe $2 + 3i$:

" « Hello World » en Python dans un interpréteur interactif"

```
>>> 2+3j
(2+3j)
>>> type(2+3j)
<class 'complex'>
```

Revenons maintenant sur les flottants. l'interpréteur interactif peut être utilisé comme une calculette. Pour l'instant nous ne parlerons que des opérations élémentaires +,-,* et / auxquelles nous ajouterons le quotient de la division euclidienne noté et le reste de la division euclidienne noté %. Voici quelques exemples de calculs :

" « Hello World » en Python dans un interpréteur interactif"

```
>>> 3.14*2+1
7.28
>>> 3.14+5+2j
(8.14+2j)
```

```
>>> (1+1j)+(2-1j)
(3+0j)
```

Vous pouvez voir que la priorité des opérateurs est respectée et que les opérations entre nombres de typés différents sont gérées (par exemple l'ajout d'un flottant et d'un complexe produit un complexe).

Au niveau du traitement de la division et de la division euclidienne, nous allons pouvoir noter des différences intéressantes entre Python 2.7 et Python 3.2. En effet, voici un exemple de tests de divisions en Python 2.7 :

Division avec Python2.7

```
>>> 7/3
2
>>> 7./3.
2.3333333333333335
>>> 7//3
2
>>> 7%3
1
```

Les résultats dépendent du contexte ! Dans un contexte entier, la division donne pour résultat le quotient de la division euclidienne alors qu'elle se comporte normalement dans un contexte flottant. Effectuons les mêmes tests en Python 3.2 :

Division avec Python3.2

```
>>> 7/3
2.3333333333333335
>>> 7.0/3.
2.3333333333333335
>>> 7//3
2
>>> 7%3
1
```

Les résultats paraissent ici logiques : 7/3 et 7.0/3.0 donnent le même résultat sous la forme d'un flottant et 7//3 a pour résultat 2 qui est bien le quotient de la division euclidienne de 7 par 3.

Si vous utilisez les calculs en Python, l'usage de la division peut donc être source d'erreur et il faudra donc bien se souvenir de la version de Python employée. Mais il y a pire ... et ce n'est pas l'apanage de Python. Le phénomène que je m'apprête à souligner est vrai pour absolument tous les langages, mais il est bien souvent oublié. Quel est le résultat d'une opération simple telle que ***((0.7 + 0.1)*10)*** ?

Vous pensez que le résultat est 8 ? Testons donc en Python :

Listing –

```
>>> ((0.7+0.1)*10)
7.999999999999999
```

On dirait qu'il y a un problème ... En fait cela provient de la représentation des flottants en machine : ils sont arrondis [3] ! En Python, la solution consistera a utiliser un module spécial, le module ***Decimal*** [4].

Comme nous n'avons pas encore vu comment utiliser les modules, je ne m'étendrai pas sur ce problème, mais si vous manipulez des flottants pour des calculs précis, méfiez vous !

En dehors des opérateurs élémentaires, les éléments numériques peuvent être assignés à des variables a l'aide du signe =(***variable = valeur***) et il est possible d'effectuer une opération en utilisant l'ancienne valeur d'une variable à l'aide de *+=*, *-=*, **=* et */=* (***variable += valeur*** équivaut à ***variable = variable + valeur***).

Voici un petit exemple :

Listing –

```
>>> a+=3
>>> a=2
>>> a
2
>>> a+=3
>>> a
5
```

Les opérateurs de pré- et de post- incrémentation ou décrémentation *++* ou *−* n'existent pas en Python.

Par contre, vous avez la possibilité d'effectuer des affectations de variables multiples avec une même valeur :

Listing –

```
>>> a=b=1
>>> a
1
>>> b
1
```

Et vous pouvez également effectuer des affectations multiples de variables ayant des valeurs différentes en séparant les noms de variables et les valeurs par des virgules :

Listing –

```
>>> a, b=2, 3
>>> a
2
>>> b
3
```

Ce mécanisme peut paraître anodin, mais il permet de réaliser très simplement une permutation de valeurs sans avoir recours a une variable intermédiaire :

Listing –

```
>>> a=2
>>> b=3
>>> a,b=b,a
>>> a
3
>>> b
2
```

Enfin, un opérateur intéressant pour enlever un nombre a une puissance quelconque : l'opérateur ***.

Listing –

```
>>> 2**3
8
```

Après avoir vu les types numériques, voyons maintenant comment stocker des chaînes de caractères.

1.2.2 Les chaîne caractères

Les chaînes de caractères sont encadrées par des apostrophes ou des guillemets. Le choix de l'un ou de l'autre n'a aucun impact au niveau des performances. La concaténation (assemblage de plusieurs chaînes pour n'en produire qu'une seule) se fait à l'aide de l'opérateur *+*. Enfin, l'opérateur ***** permet de répéter une chaîne. Voyons cela sous forme d'exemples :

Listing –

```
>>> a="Hello "
>>> b='World !'
>>> a+b
'Hello World !'
>>> 3*a
'Hello Hello Hello '
```

L'acces à un caractère particulier de la chaîne se fait en indiquant sa position entre crochets (le premier caractère est à la position 0) :

Listing –

```
>>> a[0]
'H'
```

En Python, on peut faire beaucoup de choses avec les chaînes de caractères ... nous verrons cela plus en détail dans l'article consacré aux chaînes et aux listes.

1.2.3 Les boléens

Les valeurs booléennes sont notées ***True*** et ***False*** (attention à la majuscule). Dans le cadre d'un test, les valeurs ***0***, ***""*** (la chaîne vide) et ***None*** sont considérées comme étant égales à ***False***. Ces valeurs sont retournées comme résultats des tests réalisés à l'aide des opérateurs de comparaison : *==* pour l'égalité, ***!=*** pour la différence, puis *<*, *>*, *<=* , et *>=*. Exemple :

Listing –

```
>>> 2==3
False
>>> 2<=3
True
```

Pour combiner les tests, on utilise les opérateurs booléens : ***and*** (et), ***or*** (ou) et ***not*** (non). Voici des exemples de combinaisons de tests :

Listing –

```
>>> 0 and True
0
>>> 1 and True
True
>>> not(0 or True)
False
```

1.2.4 Les listes, les tuples et les dictionnaires

Nous allons maintenant voir trois types particuliers présentant de fortes similitudes : les listes, les tuples et les dictionnaires. Ces types étant un peu complexes, il ne s'agit ici que d'un premier aperçu, nous les étudierons plus en détail dans l'article leur étant consacré.

Les listes

Une liste est un ensemble d'éléments éventuellement de différents types. Une liste se définit à laide des crochets et une liste vide est symbolisée par *[]*.

L'accès à un élément de la liste se fait en donnant son index, de la même manière qu'avec les chaînes de caractères :

Listing –

```
>>> maliste=[3.14, "cours isn", 1+1j, 5]
>>> maliste[1]
'cours isn'
```

Les tuples

Les tuples sont des listes particulières dont nous verrons l'intérêt plus tard. Ils sont définis par des parenthèses et leurs éléments sont accessibles de la même manière que pour les listes. Notez que pour lever toute ambiguïté, un tuple ne contenant qu'un seul élément sera noté (élément,). Si vous omettez la virgule, Python pensera qu'il s'agit d'un parenthésage superflu et vous n'aurez donc pas créé un tuple. L'exemple des listes peut tout a fait être rapporté aux tuples :

Listing –

```
>>> montuple=(3.14, "cours isn", 1+1j, 5)
>>> montuple[1]
'cours isn'
```

Les dictionnaires

Pour en finir avec notre aperçu des types complexes, voici les dictionnaires.

Un dictionnaire est une liste où l'accès aux éléments se fait à l'aide d'une clé alphanumérique ou purement numérique. Il s'agit d'une association clé/valeur sous la forme ***clé:valeur***. Les dictionnaires sont définis a l'aide des accolades. Dans d'autres langages les dictionnaires sont aussi appelés tableaux associatifs ou tables de hachage. Voici un exemple d'utilisation d'un dictionnaire :

Listing –

```
>>> t ={'nom':'toto', 'prenom':'titi'}
>>> t['nom']
'toto'
```

Ce type de données conclut notre tour des types de données en Python.

Comment articuler maintenant plusieurs instructions pour écrire un programme ?

1.3 Le syntaxe générale

Dans tous les langages, l'indentation du code (c'est-a-dire la manière d'écrire le code en laissant des espaces de décalage en début des lignes) est laissée au choix éclairé du développeur. Mais, force est de le constater, parfois le développeur est un sombre idiot... Qui n'a jamais vu de code ressemblant au programme suivant (exemple en PHP) ?

Exemple en PHP

```
if ($a== 1){
        echo 'salut !';}
else {
echo 'Au revoir';}
```

Ce code peut aussi être écrit différemment :

Exemple en PHP

```
if ($a== 1)
echo 'salut !';
else echo 'Au revoir';
```

Python oblige donc le développeur à structurer son code a l'aide des indentations : ce sont elles qui détermineront les blocs (séquences d'instructions liées) et non les accolades comme dans la majorité des langages. Les blocs de code sont déterminés par :

- la présence du caractère **:** en fin de ligne ;
- une indentation des lignes suivantes à l'aide de tabulations ou d'espaces (attention a ne pas mélanger les deux, Python n'aime pas ça du tout : votre code ne fonctionnera pas et vous n'obtiendre pas de message d'erreur explicite).

Enfin, il n'y a pas de point-virgule en fin de ligne. Même si ce caractère peut-être utilisé pour séparer des instructions sur une même ligne, son usage est déconseillé, Le caractère # permet d'indiquer que tous les caractères suivants sur la même ligne ne doivent pas être interprétés. il s'agit de commentaires

Exemple en PHP

```
>>> a = 12 # Affectation d'un entier
```

Pour pouvoir créer réellement un script, il nous faut des structures permettant d'effectuer des tests et des traitements répétitifs. C'est ce que nous allons voir dans la suite.

1.4 Les structures de boucle et de test

Nous séparerons ici les tests et les boucles.

1.4.1 Les structures de test

Il n'existe en fait qu'une seule structure de test en Python : le test « si alors » utilisant les instructions ***if*** et ***else***. Au niveau de la construction du test, il faudra vérifier une condition et si cette dernière est vraie exécuter un bloc (et donc penser aux deux-points en fin de ligne et a l'indentation). La structure d'un test est donc :

Exemple en PHP

```
if condition:
        # Traitement bloc 1
else:
        # Traitement bloc 2
```

Pour les tests multiples, il faudra enchaîner les ***if*** en cascade grâce à l'instruction ***elif***, contraction de *else if :*

Exemple en PHP

```
if condition:
        # Traitement bloc 1
elif:
        # Traitement bloc 2
else:
        # Traitement bloc 3
```

Sur un exemple concret cela donne :

Exemple en PHP

```
>>> a=3
>>> if a>1:
...     print('a>1')
... else:
...     print('a<=2')
...
a>1
```

Notez qu'en mode interactif, Python affiche des points de suspension pour indiquer que votre commande est incomplète car un bloc a été ouvert mais pas fermé. Il faut appuyer sur ***<Entrée>*** dans la dernière ligne pour fermer le bloc.

1.4.2 Les structures de boucle

Pour les boucles il existe deux structures. La première est le ***for*** qui prend ses valeurs dans une liste : ***for variable in liste_valeurs***. À chaque itération la variable prendra pour valeur un élément de la liste. Voici un exemple :

Exemple en PHP

```
>>> for e in ['pomme','poire','kiwi']:
...     print(e)
...
pomme
```

```
poire
kiwi
```

Pour créer une boucle équivalente aux boucles traditionnelles « pour i variant de n à m, faire... », nous utiliserons la fonction range

En utilisant cette fonction, on peut donc mimer le comportement d'une boucle for traditionnelle avec une variable qui prendra ses valeurs entre deux entiers donnés :

Exemple en PHP

```
>>> for i in range(0,4):
...     print(i)
...
0
1
2
3
```

Le mot-clé ***continue*** permet de passer à l'itération suivante et le mot-clé ***break*** permet de sortir de la boucle :

Exemple en PHP

```
>>> for i in range(0,10):
...     if i==2:
...         continue
...     if i==4:
...         break
...     print(i)
...
0
1
3
```

La seconde structure de boucle est le « tant que » : tant qu'une condition est vraie, on boucle. Attention : dans ce type de boucle on utilise une variable de boucle dont la valeur doit changer pour pouvoir sortir de la boucle :

Exemple en PHP

```
>>> while i<4:
...     print(i)
...     i+=1 #équivalent à i=i+1
...
0
1
2
3
```

La boucle ***while*** accepte elle aussi les mots-clés ***continue*** et ***break***.

1.5 La structure d'un programme

Avec L'ensemble des instructions que nous avons vu, vous pouvez écrire des programmes plus longs que les quelques lignes testées dans l'interpréteur interactif. Vous allez donc avoir besoin de stocker vos scripts dans des fichiers d'extension ***.py***.

Pour exécuter ces fichiers de code vous aurez alors deux possibilités :

- soit lancer simplement dans un terminal :

```
python monFichier.py
```

- soit ajouter à votre code source une ligne indiquant où se trouve l'interpréteur Python à utiliser :

```
#!/usr/bin/python
```

Cette ligne doit être la premièreccligne de votre fichier. Ensuite vous n'avez plus qu'a rendre votre fichier exécutable (***chmod u+x monFichie.py***) et vous pourrez lancer votre script par :

```
monFichier.py
```

1.5.1 Conclusion

Cet article a permis d'avoir un aperçu global de ce qu'était Python. Avec les connaissances acquises vous devriez être capable de réaliser déjà de petits scripts. Les articles suivants vont vous permettre d'approfondir vos connaissances sur des points précis du langage.

Chapitre 2

Le slicing et les structures de liste

Cet article constitue un approfondissement sur le fonctionnement des listes, des tuples et des dictionnaires. Nous apporterons une attention particulière à la technique du « slicing » ou « découpage en tranches ».

Dans les articles précédents vous avez pu vous familiariser avec différents types de données Python. Nous avons notamment vu des structures complexes permettant de stocker d'autres variables : les listes et les tuples. Je vous propose d'approfondir vos connaissances sur ces structures et d'étudier la technique du « slicing » qui permet de spécifier de manière simple des plages de données que l'on souhaite utiliser. Pour commencer, revenons sur une liste particulière :

2.1 la chaîne de caractères.

Une chaîne de caractères est une liste particulière ne contenant que des caractères. Comme pour une liste, on peut accéder à ses éléments (les caractères) en spécifiant un indice :

```
In [1]: chaine= 'Lycée Pablo Neruda'
In [2]: chaine[0]
Out[2]: 'L'
```

Par contre, il est impossible de modifier une chaîne de caractères ! On dit alors qu'il s'agit d'une liste non modifiable :

```
In [3]: chaine[1]='a'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-47fe7d6fd6ed> in <module>()
----> 1 chaine[1]='a'
TypeError: 'str' object does not support item assignment
```

Vous pourriez avoir la sensation de modifier une chaîne de caractères si vous décidiez de lui ajouter des caractères en fin de chaîne à l'aide d'une concaténation du type suivant :

```
In [4]: chaine=chaine+'!'
```

En fait, ici, nous réalisons une nouvelle affectation : la variable ***chaine*** est écrasée (effacée puis remplacée) par la valeur ***chaine+'!'***. l| ne s'agit pas d'une modification au sens propre du terme.

Les chaînes de caractères admettent une écriture particulière appelée « formatage ». La syntaxe à employer est légèrement différente qu'il s'agisse de Python 2.7 ou Python 3.2. Le formatage consiste à écrire une chaîne de caractères en y insérant des variables dont nous donnerons la valeur par la suite. On peut voir ce mécanisme comme dans un texte à trous pour lequel la réponse serait donnée a posteriori. En Python 2.7 on utilisera les expressions de formatage issues du C :

- ***%d*** pour désigner un entier :
- ***%f*** pour un flottant ;
- ***%s*** pour une chaîne de caractères ;

- ***%.4f*** pour forcer l'affichage de 4 chiffres après la virgule ;
- ***%02d*** pour forcer l'affichage d'un entier sur deux caractères ;
- etc ...

Vous trouverez une liste complète des expressions reconnues sur la page :
`https/ldocspythonorg/library/stringhtml`.
Pour utiliser ce formatage, il faudra donner une chaîne de caractères contenant des expressions de formatage et faire suivre cette chaîne du caractère % et d'un tuple contenant les valeurs de remplacement :

```
>>> c=2
>>> d=3
>>> '%d*%d=%d'%(c,d,c*d)
'2*3=6'
```

En Python 3.2, les indications de formatage sont données par ***{}*** sans indication de type. Il est possible d'utiliser les expressions vues précédemment mais en utilisant le caractère : à la place de %. Cette fois, le formatage est réalisé par la méthode ***format*** qui s'applique à une chaîne de caractères :

```
>>> '{}*{}={}'.format(c,d,c*d)
'2*3=6'
>>> chaine='un entier sur deux chiffres: {:02d}'
>>> chaine.format(6)
'un entier sur deux chiffres: 06'
```

2.2 Les spécificités des tuples

Les tuples sont également des listes non modifiables (les chaînes de caractères sont donc en fait des tuples particuliers) :

```
>>> t=(1,2,3)
>>> t[0]
1
>>> t[0]=0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Attention de ne pas utiliser le mot-clé tuple comme nom de variable : ce dernier permet de créer un tuple de manière explicite. Il vous faudra utiliser en paramètre... une liste. Il permet donc d'effectuer une conversion :

```
>>> l=[1,2,3]
>>> t=tuple(l)
>>> t
(1, 2, 3)
>>> l
[1, 2, 3]
>>> t=tuple('1234')
```

```
>>> t
('1', '2', '3', '4')
```

Tout comme avec les chaînes, il sera possible de concaténer des tuples et avoir la sensation d'avoir modifié un tuple alors que nous aurons simplement réaffecté une variable :

```
>>> t=(1,2,3)
>>> t=t+(4,5,6)
>>> t
(1, 2, 3, 4, 5, 6)
```

Pour rappel, un tuple ne contenant qu'un seul élément est noté entre parenthèses, mais avec une virgule précédant la dernière parenthèse :

Quel peut être l'intérêt d'utiliser ce type plutôt qu'une liste ?

Tout d'abord, les données ne peuvent pas être modifiées par erreur mais surtout, leur implémentation en machine fait qu'elles occupent moins d'espace mémoire et que leur traitement par l'interpréteur est plus rapide que pour des valeurs identiques mais stockées sous forme de listes.

De plus, de par leur aspect non modifiable, les valeurs contenues dans un tuple peuvent être utilisées en tant que clé pour accéder à une valeur dans un dictionnaire (alors que c'est beaucoup plus dangereux avec les valeurs contenues dans une liste) :

```
>>> l=['cle1','cle2','cle3']
>>> t=('cle1','cle2','cle3')
>>> d={'cle1':1,'cle2':2,'cle3':3}
>>> d[l[0]]
1
>>> l[0]='cle_modifiee'
>>> d[l[0]]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'cle_modifiee'
```

2.3 Les spécificités des listes

Nous avons vu que les listes étaient des éléments modifiables pouvant contenir différents types de données.

Il est bien sûr possible de modifier les éléments d'une liste :

```
>>> l=[1,2,3,4]
>>> l
[1, 2, 3, 4]
>>> l[0]=0
>>> l
[0, 2, 3, 4]
```

Plusieurs méthodes peuvent être employées pour ajouter des éléments a une liste existante (sans réaffectation). La méthode ***append()*** ajoute un élément en fin de liste et la méthode ***insert()*** permet de spécifier l'indice où insérer un élément :

```
>>> l.append(5)
>>> l
[0, 2, 3, 4, 5]
>>> l.insert(1,1)
>>> l
[0, 1, 2, 3, 4, 5]
```

La méthode ***extend()*** permet de réaliser la concaténation de deux listes sans réaffectation. Cette opération est réalisable avec réaffectation en utilisant l'opérateur *+* :

```
>>> l1=[1,2,3]
>>> l2=[4,5,6]
>>> l1.extend(l2)
>>> l1
[1, 2, 3, 4, 5, 6]
>>> l1=[1,2,3]
>>> l1=l1+l2
>>> l1
[1, 2, 3, 4, 5, 6]
```

Pour supprimer un élément, on peut utiliser la méthode ***remove()*** qui supprime la première occurrence de la valeur passée en paramètre ou la commande Ce] qui supprime un élément précis en fonction de son indice :

```
>>> l=[1,2,3,1]
>>> l.remove(1)
>>> l
[2, 3, 1]
>>> l.remove(1)
>>> l
[2, 3]
>>> l.remove(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
>>> l=[1,2,3,1]
>>> del l[3]
>>> l
[1, 2, 3]
```

Pour savoir si un élément appartient bien à une liste on utilise le mot-clé ***in***. Si l'élément testé est dans la liste, la valeur retournée sera ***True*** et sinon ce sera ***False*** :

```
>>> distrib = [ 'debian', 'ubuntu', 'fedora']
>>> 'debian' in distrib
True
>>> 'mandriva' in distrib
False
```

Il est également possible d'obtenir l'indice de la première occurrence d'un élément par la méthode ***index()*** :

```
>>> distrib.index('ubuntu')
1
```

Pour connaître la taille d'une liste (nombre d'éléments qu'elle contient), on pourra utiliser la fonction ***len()*** :

Mais attention : cette fonction ne compte que les éléments de « premier niveau » ! Si vous avez une structure de liste complexe contenant des sous-listes, chaque sous-liste ne comptera que comme un seul élément :

```
>>> l=[1,2,['a','b',['000','001','010']]]
>>> len(l)
3
```

Ce problème du décompte des sous-listes nous amènera à un autre problème : celui de la copie des listes que nous aborderons après le slicing.

l'accès aux éléments d'une sous » liste s'effectue en spécifiant l'indice de la sous—liste suivi de l'indice de l'élément souhaité dans la sous-liste, etc. En reprenant la liste utilisée dans l'exemple précédent :

```
>>> l=[1,2,['a','b',['000','001','010']]]
>>> len(l)
3
>>> l[0]
1
>>> l[2]
['a', 'b', ['000', '001', '010']]
>>> l[2][0]
'a'
>>> l[2][2]
['000', '001', '010']
>>> l[2][2][1]
'001'
```

Pour compléter cet approfondissement sur les listes, sachez que Python implémente un mécanisme appelé « compréhension de listes », permettant d'utiliser une fonction qui sera appliquée sur chacun des éléments d'une liste. Voici un exemple :

```
>>> l=[0,1,2,3,4,5]
>>> carre=[x**2 for x in l]
>>> carre
[0, 1, 4, 9, 16, 25]
```

À l'aide de l'instruction ***for x in l***, on récupère un à un les éléments contenus dans la liste ***l*** et on les place dans la variable ***x***. On élève ensuite chacune des valeurs au carré (***x**2***) dans un contexte de liste, ce qui produit une nouvelle liste. Ce mécanisme peut produire des résultats plus complexes. Il peut également être utilisé avec les dictionnaires.

2.4 Les spécificités des dictionnaires

Comme nous l'avons vu, les dictionnaires sont des listes d'éléments indices par des clés. Comme pour les listes, la commande de ***del*** permet de supprimer un élément et son mot clé , « in » permet de vérifier [existence d'une clé :

```
>>> d={'cle1':1, 'cle2':2, 'cle3':3}
>>> del d['cle2']
>>> d
{'cle3': 3, 'cle1': 1}
>>> 'cle1' in d
True
>>> 'cle2' in d
False
```

Vous aurez pu remarquer lors de l'affichage du dictionnaire ***d*** que les couples clé/valeur n'étaient pas affichés dans l'ordre de création. C'est tout à fait normal car les dictionnaires ne sont pas ordonnés !

Plusieurs méthodes permettent de récupérer des listes de clés, de valeurs et de couples clé/valeur :

```
>> courses ={ 'pommes':3, 'poires':5 , 'kiwis':7}
>>> courses.keys()
dict_keys(['poires', 'kiwis', 'pommes'])
>>> courses.values()
dict_values([5, 7, 3])
>>> courses.items()
dict_items([('poires', 5), ('kiwis', 7), ('pommes', 3)])
```

Ces méthodes renvoient des objets itérables : ce ne sont pas des listes et ils ne consomment donc pas autant de place mémoire que pour stocker une liste, on ne stocke qu'un pointeur vers l'élément courant. Avec ces structures vous pourrez toujours utiliser les boucles ***for*** classiques, par contre vous n'aurez plus un accès direct aux valeurs en spécifiant leur indice.

Dans un dictionnaire, pour obtenir la valeur correspondant à une clé on peut bien entendu utiliser la notation classique ***dictionnaire[clé]***, mais on peut aussi utiliser la méthode ***get()*** qui renvoie la valeur associée a la clé passée en premier paramètre ou, si elle n'existe pas, la valeur passée en second paramétré :

```
>>> courses ={ 'pommes':3, 'poires':5 , 'kiwis':7}
>>> courses.get('pommes', 'stock epuise')
3
>>> courses.get('salades', 'stock epuise')
'stock epuise'
```

On peut fusionner deux dictionnaires avec la méthode ***update()*** : Notez que si une clé existe déjà, la valeur stockée sera écrasée :

```
>>> courses ={ 'pommes':3, 'poires':5 , 'kiwis':7}
>>> courses2={'kiwis':3, 'salades':2}
>>> courses.update(courses2)
>>> courses
{'poires': 5, 'salades': 2, 'kiwis': 3, 'pommes': 3}
```

Enfin, en ce qui concerné la copie de dictionnaires, le problème sera ie même qu'avec les listes et il sera expliqué plus loin dans cet article.

2.5 Le slicing

Le ***slicing*** est une méthode applicable à tous les objets de type liste ordonnée (donc pas aux dictionnaires). ll s'agit d'un « découpage en tranches » des éléments d'une liste de manière à récupérer des objets respectant cette découpe.

Pour cela, nous devrons spécifier l'indice de l'élément de départ, l'indice de l'élément d'arrivée (qui ne sera pas compris dans la plage) et le pas de déplacement. Pour une variable ***v*** donnée, l'écriture se fera en utilisant la notation entre crochets et en séparant chacun des paramètres par le caractère deux-points : ***v[début :fin :pas]***. Cette écriture peut se traduire par : « les caractères de la variable ***v*** depuis l'indice ***début*** jusqu'à l'indice ***fin*** non compris avec un déplacement ***de*** pas caractère(s) ».

Pour bien comprendre le fonctionnement du slicing, nous commencerons par l'appliquer aux chaînes de caractères avant de voir les listes et les tuples.

2.5.1 Le slicing sur une chaîne de caratère

Voici un premier exemple simple, illustre par la figure 1 :

```
>>> c='Linux Pratique'
>>> c[:5]
'Linux'
>>> c[6:14]
'Pratique'
```

Vous avez remarque que dans ce code aucune indication de pas n'a été donnée : c'est la valeur par défaut qui est alors utilisée, c'est-a-dire ***l***. De même, si la valeur de début est omise, la valeur par défaut utilisée sera ***0*** et si la valeur de fin est omise, la valeur par défaut utilisée sera la taille de la chaîne +1 (il ne faut pas oublier que l'indice de fin indique la première lettre qui ne sera pas affichée).

```
>>> c[:5] # équivaut à c[0:5], équivaut à c[0:5:1]
'Linux'
>>> c[6:] # équivaut à c[6:14], équivaut à c[6:14:1]
'Pratique'
```

Fig. 1 : Slicing sur la chaîne c='Linux Pratique'

Du fait de ces valeurs par défaut, que pensez vous que l'on sélectionne en tapant : ***c[:]*** ou encore ***c[: :]*** ? La chaîne entière, bien sûr :

```
>>> c[:]
'Linux Pratique'
```

Il devient alors très simple d'inverser une chaîne en utilisant le pas :

```
>>> c[::-1]
'euqitarP xuniL'
```

Une première solution pour effectuer une copie peut être d'utiliser le slicing. En effet, l'opération [z] renvoie une nouvelle liste, ce qui résout le problème des pointeurs vers la même zone mémoire, comme le montrent l'exemple suivant :

```
>>> listea =[1, 2, 3]
>>> listeb =listea[:]
>>> listea
[1, 2, 3]
>>> listeb
[1, 2, 3]
>>> listeb[0]=0
>>> listeb
[0, 2, 3]
>>> listea
[1, 2, 3]
```

Cette copie peut paraître satisfaisante ... et elle l'est, mais à condition de ne manipuler que des listes de premier niveau ne comportant aucune sous-liste :

```
>>> listea=[1,2,3,[4,5,6]]
>>> listeb =listea[:]
>>> listea
[1, 2, 3, [4, 5, 6]]
>>> listeb
[1, 2, 3, [4, 5, 6]]
>>> listeb[3][0]=0
>>> listeb
[1, 2, 3, [0, 5, 6]]
>>> listea
[1, 2, 3, [0, 5, 6]]
```

En effet, le slicing n'effectue pas de copie récursive : en cas de sous-liste, on retombe dans la problématique des pointeurs mémoire. La solution est alors d'utiliser un module spécifique, le module ***copy***, qui permet d'effectuer une copie récursive. Bien que n'ayant pas encore approfondi la manipulation des modules, voici comment réaliser une copie de liste par cette méthode :

```
>>> from copy   import *
>>> listea=[1,2,3,[4,5,6]]
>>> listeb=deepcopy(listea)
>>> listea
[1, 2, 3, [4, 5, 6]]
>>> listeb
[1, 2, 3, [4, 5, 6]]
>>> listeb[3][0]=0
>>> listeb
[1, 2, 3, [0, 5, 6]]
>>> listea
[1, 2, 3, [4, 5, 6]]
```

Le problème est exactement le même avec la copie de dictionnaires. Il faudra utiliser ici la méthode ***copy()*** :

```
>>> dicoa={'cle1':1, 'cle2':2, 'cle3':3}
>>> dicob=dicoa.copy()
>>> dicoa
{'cle2': 2, 'cle3': 3, 'cle1': 1}
>>> dicob
{'cle2': 2, 'cle3': 3, 'cle1': 1}
>>> dicob['cle1']=0
>>> dicob
{'cle2': 2, 'cle3': 3, 'cle1': 0}
>>> dicoa
{'cle2': 2, 'cle3': 3, 'cle1': 1}
```

Conclusion

Les structures de liste en Python sont particulièrement fournies et il existe pour chacune d'elles de nombreuses méthodes permettant de manipuler leurs éléments. Le slicing représente un raccourci efficace pour récupérer ou modifier des éléments d'une liste : il masque la complexité des opérations sous une syntaxe claire et lisible. . . pour peu que l'on ait pris le temps de se familiariser avec elle.

Chapitre 3

LES FONCTIONS ET LES MODULES

Jusqu'à présent nous n'avons que modérément utilisé les fonctions. De plus, ces fonctions étaient déjà définies dans le langage. Dans cet article nous allons découvrir et approfondir quelques fonctions prédéfinies dé base, puis nous verrons la conception de fonctions originales et enfin, nous aborderons l'organisation de fonctions en bibliothèques ou, en Python, modules.

3.1 Fonctions prédéfinies

La toute première fonction que nous avons utilisée est la fonction ***print()*** permettant d'afficher du texte a l'écran.

3.1.1 Afficher un texte avec print

La fonction ***print()*** peut prendre plusieurs paramètres.

```
>>> print('Monsieur','Meilland')
Monsieur Meilland
```

Il est possible d'utiliser les paramètres ***set*** et ***and*** qui permettent de définir le caractère utilisé pour séparer les chaînes et de définir le caractère de fin de ligne. Ces paramètres sont utilisés d'une manière un peu particulière en spécifiant leur nom suivi du signe égal et de leur valeur lors de l'appel de la fonction.

Le mécanisme mis en jeu lors de cet appel sera analysé dans la partie consacrée aux fonctions originales. Voici quelques exemples utilisant ces paramètres qui doivent être placés en dernière position :

```
>>> print('Monsieur','Meilland', sep='**')
Monsieur**Meilland
>>> print('Monsieur','Meilland', sep=',')
Monsieur,Meilland
>>> print('Monsieur','Meilland', sep='-',end='!\n')
Monsieur-Meilland!
```

Après avoir vu comment afficher du texte, il peut être intéressant de voir la fonction complémentaire permettant à l'utilisateur de saisir des données.

Le dialogue entre l'utilisateur et le programme est important car il permet d'obtenir une interface permettant de paramétrer le comportement du programme (abstraction faite du passage de paramètres en lignes de commandes que nous verrons dans un autre article).

Les fonctions de la famille ***input*** permettent d'afficher un message à l'écran et de donner la main à l'utilisateur pour qu'il saisisse des caractères au clavier et qu'il appuie sur la touche [Entrée] pour valider.

```
>>> c=input('Veuillez saisir un entier:')
Veuillez saisir un entier:2
>>> c
```

```
'2'
>>> c=input('Veuillez saisir une chaine: ')
Veuillez saisir une chaine: coucou
>>> c
'coucou'
```

Les principales fonctions de conversion sont :

- ***int()*** pour convertir en entier ;
- ***float()*** pour convertir en flottant ;
- ***complexe()*** pour convertir en complexe ;
- ***str()*** pour convertir sous forme de chaîne de caractères (inutile dans le cadre d'une utilisation avec ***input()***).

3.1.2 Recherche de l'aide avec help()

On ne peut pas connaître par coeur la syntaxe de toutes les commandes ! Il est donc nécessaire de savoir où chercher des informations. La fonction ***help()*** permet d'avoir accès à une aide (minimaliste) relative à la fonction passée en paramètre. De manière générale, le nom de la fonction pour laquelle on recherche de l'aide doit être donné sans guillemets alors que les commandes doivent être sous forme de chaînes de caractères (encadrées par des guillemets ou des apostrophes). Après affichage de l'aide, tapez sur la touche <q> pour revenir au prompt de l'éditeur interactif :

```
>>> help(input)
Help on built-in function input in module builtins:

input(...)
    input([prompt]) -> string
    Read a string from standard input.  The trailing newline is stripped.
```

Vous aurez accès a cette aide depuis le shell en utilisant la commande ***pydoc*** :

```
meilland@debian:~\$ pydoc  input
```

Comme en mathématiques, une fonction effectue un calcul, un traitement, puis renvoie un résultat. Une fonction a un nom et accepte ou non des paramètres. Prenons Fexemple de la fonction mathématique $f(x) = x^2$. En Python, cela donnerait :

```
def f(x):
        return x**2
```

En ligne 1 on définit l'en-tête de la fonction a l'aide du mot clé def : la fonction s'appelle f et elle admet un paramètre noté x, La ligne se termine par le caractère deux-points :, on ouvre un nouveau bloc, toutes les lignes suivantes qui seront indentées feront partie des instructions exécutées lors de l'appel à la fonction.

La ligne 2 indique la valeur a renvoyer à l'aide du mot-clé ***return***. Cette fonction pourra alors être appelée :

```
>>> print(f(2))
4
```

Si vous omettez de préciser une valeur de retour pour une fonction, Python renverra automatiquement la valeur ***None***.

Vous avez bien sûr la possibilité de créer des fonctions acceptant aucun ou de nombreux paramètre(s) :

```
>>> def hello():
...     print('bonjour!')
...
>>> hello()
bonjour!
>>> def maFct(a,b,c):
...     print('Paramètre: {}, {} et {}'.format(a,b,c))
...
>>> maFct(1,2,'coucou')
Paramètre: 1, 2 et coucou
```

La documentation relative à une fonction s'écrit à l'aide d'une chaîne simple située sous l'entête de la fonction. Si vous avez besoin de plus d'une ligne, vous pouvez utiliser les commentaires multi-lignes qui sont encadrés par des triples guillemets. Les commentaires ajoutés ainsi à une fonction sont ensuite accessibles par la fonction ***help(:)***

```
>>> def ma_fonction():
...     '''Fonction affichant un message
...     valeur de retour: None'''
...     print('coucou')
help(ma_fonction)
Help on function ma_fonction in module __main__:

ma_fonction()
    Fonction affichant un message
    valeur de retour: None
(END)
```

De plus, Python dispose de mécanismes permettant de manipuler très simplement les paramètres des fonctions.

Il est possible de définir des paramètres ayant une valeur par défaut. Si ces paramètres ne sont pas spécifiés lors de l'appel à la fonction, ce seront les valeurs par défaut qui seront utilisées. La seule contrainte est que les paramètres ayant une valeur par défaut doivent être positionnés a la fin de la liste des paramètres :

```
def ma_fonction(a, b, c=2, d=3):
```

Cette fonction pourra être appelée de trois façons différentes :

```
>>> ma_fonction(0,1)     #a=0, b=1, c=2, d=3
>>> ma_fonction(0,1,4)   #a=0, b=1, c=4, d=3
>>> ma_fonction(0,1,4,5) #a=0, b=1, c=4, d=5
```

3.1.3 Ordre des paramètres

En Python, l'ordre des paramètres d'une fonction n'est pas fixé si l'on est capable de les nommer. Nous avons vu un exemple de hommage de paramètres précédemment avec la fonction ***print()*** et les paramètres ***sep*** et ***end***. Voici un exemple d'application de ce concept :

```
>>> def premier(a,b):
...     return a
...
>>> premier(b=5,a=2)
```

Bien entendu, il est encore plus recommandé qu'à l'accoutumée d'utiliser des noms de paramètres « parlants » et pas seulement des lettres comme dans l'exemple précédent.

3.1.4 Nombre de paramètres non fixé

Un mécanisme très utile de Python est basé sur le fait de pouvoir créer une fonction sans fixer au préalable le nombre de ses paramètres. On utilise pour cela les ****args*** et les *****kargs*** :

- ****args*** : tuple contenant la liste des paramètres passés par l'utilisateur ;
- *****kargs*** : dictionnaire des paramètres.

La seule différence entre les deux syntaxes (******* ou ********) devant le nom du paramètre) est le type de la variable de retour : un tuple dans le premier cas et un dictionnaire dans le deuxième cas (les paramètres doivent alors être saisis sous la forme ***nom=valeur***). Voici un exemple d'application :

```
>>> def exemple(*args):
...     print(args)
...
>>> exemple(1,2,"coucou")
(1, 2, 'coucou')
>>> def exemple2(**kargs):
...     print(kargs)
...
>>> exemple2(i=1,j=2,text="coucou")
{'i': 1, 'text': 'coucou', 'j': 2}
```

Il est possible de mixer les écritures de paramètres fixés et non fixes. Ceci permet disposer des paramètres et d'en rendre d'autres optionnels :

```
def affiche_style(text, **options):
        style=""
        if 'color' in options:
                style +=' color:'+options['color']+';\n'
        if 'font' in options:
                style +=' font:'+options['font']+';\n'
        print('style\n{\n'+style+'}\n\n'+'Texte:'+text)
affiche_style('Cours ISN', color='#f00', font='Arial 12pt')
```

L'exécution de ce code produira l'affichage suivant :

```
style
```

```
{
 color:#f00;
 font:Arial 12pt;
}

Texte:Cours ISN
```

3.1.5 Type de passage des paramètres : valeur ou adresse ?

Pour finir sur les fonctions, intéressons nous a la manière dont sont transmises les variables passées en paramètres.

Sont-elles passées par valeur, c'est à dire qu'une copie de la variable sera effectuée et que toute modification de la valeur de la variable à l'intérieur de la fonction ne sera pas conservée à la sortie de la fonction ? Ou bien sont-elles passées par adresse (ou encore par référence), c'est à dire que l'on travaille directement sur la variable et que toute modification de la variable dans la fonction est valable sur l'ensemble du programme ?

En Python, tous les paramètres sont passés par adresse ! Par contre, comme certains types ne sont pas modifiables, on aura alors l'impression qu'ils sont passés par valeur.

Les types non modifiables sont les types simples (entiers, flottants, complexes, etc.), les chaînes de caractères et les tuples.

Voici un exemple utilisant un paramètre non modifiable :

```
>>> def incremente(a):
...     a=a+1
...
>>> valeur=5
>>> valeur
5
>>> incremente(valeur)
>>> valeur
5
```

Avec un paramètre modifiable, tel qu'une liste, les modifications seront visibles en sortie de la fonction :

```
>>> def ajoute(liste,val):
...     liste.append(val)
...
>>> l=[]
>>> l
[]
>>> ajoute(l,5)
>>> l
[5]
```

Ayant bien étudié la création des fonctions, voyons maintenant comment créer des ensembles cohérents de fonctions.

3.2 Les modules