Introduction

Python est un langage de programmation puissant et facile à apprendre. Il dispose de structures de données de haut niveau et d'une approche de la programmation orientée objet simple mais efficace. Python est un langage idéal pour l'écriture de scripts et le développement rapide d'applications dans de nombreux domaines et sur de nombreuses plateformes.

Nombres, Types et Chaines de caractères

1. Nombres

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5.0*6) / 4
50
>>> 8 / 5.0
1.6
```

Les nombres entiers (comme 2, 4, 20) sont de type **int**, alors que les décimaux (comme 5.0, 1.6) sont de type **float**.

```
>>> 17 / 3   # int / int -> int
5
>>> 17 /3.0   # int / float -> float
5.666666666666667
>>> 17 // 3.0  # division entière
5.0
>>> 17 % 3  # L'opérateur % retourne le reste de la division
2
>>> 5 * 3 + 2  # résultat * diviseur + reste
17
```

```
>>> 5 ** 2  # 5 carrée
25
>>> 2 ** 7  #  puissance
128
```

Opérations logiques

```
>>> x = 1   #  0001
>>> x << 2      #  Décalage du 1 à gauche de 2 bits  0100
4
>>> x | 2   #  ou  logique : 0011
3
 >>> x & 1   #   et  logique : 0001
1
```

2. Chaines de caractères

Python peut aussi manipuler des chaînes de caractères, qui peuvent être exprimés de différentes manières. Elles peuvent être écrites entre guillemets simples ('...') ou entre guillemets ("...")

```
>>> 'chaine'     #
'chaine'
>>> 'l\'eau'      #
 "l'eau"
>>> 'py' + 'thon'
'python'
>>> S = 'spam'
>>> S[0], S[2]
('s', 'a')
```

Structures de données

1. Listes

Python connaît différents types de données *combinés*, utilisés pour regrouper plusieurs valeurs. La plus souple est **la *liste***, qui peut être écrite comme une suite de valeurs (éléments) séparés par des virgules placée entre crochets. Les éléments d'une liste ne sont pas obligatoirement tous du même type.

1.1/ Définition et Déclaration

```
>>> liste1 = [1, 4, 9, 16, 25]
[1, 4, 9, 16, 25]
```

Comme les chaînes de caractères (et toute autre types de séquence), les listes peuvent être indicées et découpées :

```
>>> liste1[0]  #
1
>>> liste1[-1] #  compte negative : à partir de la droite
25
>>> liste1[-3:]  #   à partir de l'indice -3 et plus
[9, 16, 25]
```

Toutes les opérations par tranches renvoient une nouvelle liste contenant les éléments demandés. Ce qui signifie que l'opération suivante renvoie une copie superficielle de la liste :

```
>>> liste1[:]
[1, 4, 9, 16, 25]
```

Les listes gèrent aussi les opérations comme la concaténation :

```
>>> liste1 + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Mais à la différence des chaînes qui sont immuables(ne changent pas), les listes sont mutables : il est possible de changer leur contenu :

```
>>> cubes = [1, 8, 27, 65, 125]  #
>>> 4 ** 3  #
64
>>> cubes[3] = 64  # remplacer la valeur erronée
>>> cubes
[1, 8, 27, 64, 125]
```

Il est aussi possible d'ajouter de nouveaux éléments à la fin d'une liste avec la méthode **append()**. (Les méthodes seront abordées plus tard)

```
>>> cubes.append(216)  # ajouter la valeur 216 à la liste cubes
>>> cubes.append(7 ** 3)  #  ajouter le résultat de l'expression
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

Des affectations de tranches sont également possibles, ce qui peut même modifier la taille de la liste ou la vider complètement :

```
>>> lettres = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> lettres
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # remplacer quelques valeurs
>>> lettres[2:5] = ['C', 'D', 'E']
>>> lettres
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # Supprimer des valeurs
>>> lettres[2:5] = []
>>> lettres
['a', 'b', 'f', 'g']
>>> # effacer la liste
>>> lettres[:] = []
>>> lettres
[]
```

La primitive **len()** s'applique aussi bien aux chaines de caractères qu'aux listes :

```
>>> lettres = ['a', 'b', 'c', 'd']
>>> len(lettres)
4
>>> h= 'chaines'
>>> len(h)
7
```

Il est possible d'imbriquer des listes (de créer des listes contenant d'autres listes), par exemple :

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]     donner la valeur indicée par la position 1 du premier
element de la liste x
'b'
>>> x[1][2]
3
>>> x[2][2]
erreur
```

Copier une liste

```
>>> x = [7, 8, 4]
>>> y = x
>>> y
[7, 8, 4]
>>> y = x[:]
>>> y[0] = 10
>>> x
[7, 8, 4]
>>> y
[10, 8, 4]
```

Pour savoir si un élément est dans une liste, vous pouvez utiliser le mot clé **in** de cette manière:

```
>>> x = [7, 8, 4]
>>> 3 in x
False
>>> 7 in x
True
```

La fonction range génère une liste composée d'une simple suite arithmétique.

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

Quelques opérations utiles :

```
>>> liste = [1, 2, 3, 4]
>>> liste[:2]    affiche les 2 premier éléments
[1, 2]
>>> liste[-1]    affiche le dernier élément
4
>>> liste[-3]    affiche le troisième élément en partant de la fin
2
>>> liste[-2:]   affiche les 2 derniers éléments de la liste
```

```
[3, 4]
>>> liste1 = [1, 2, 3]
```

```
>>> liste2 = [4, 5]
>>> liste1 + liste2    addition de listes
[1, 2, 3, 4, 5]
>>> [0] * 4
[0, 0, 0, 0]    initialiser une liste
```

1.2/ Opérations sur les listes

Le type list dispose de méthodes supplémentaires. Voici la liste complète des méthodes des objets de type liste :

1. **`list.append(x)`** : Ajoute un élément à la fin de la liste.

2. **`list.insert(i, x)`** : Insère un élément à la position indiquée. Le premier argument est la position de l'élément courant avant lequel l'insertion doit s'effectuer, donc `a.insert(0, x)` insère l'élément x en tête de la liste a, `a.insert(len(a), x)` est équivalent à `a.append(x)`.

3. **`list.remove(x)`** : Supprime de la liste le premier élément dont la valeur est x. Une exception est levée s'il existe aucun élément avec cette valeur.

4. **`list.pop([i])`** : Enlève de la liste l'élément situé à la position indiquée, et le retourne. Si aucune position n'est indiqué, `a.pop()` enlève et retourne le dernier élément de la liste

5. **`list.clear()`** : Supprime tous les éléments de la liste, équivalent à `del a[:]`.

6. **`list.count(x)`** : Retourne le nombre d'éléments ayant la valeur x dans la liste.
7. **`list.reverse()`** : Inverse l'ordre des éléments de la liste,

Exemple:

```
>> fruits = ['orange', 'pomme', 'poire', 'banane', 'kiwi',
'pomme', 'banane']
>>> fruits.count('pomme')
2
>>> fruits.count('peiche')
0
>>> fruits.index('banane')
3
>>> fruits.index('banane', 4)  # trouver l'indice de la prochaine
banane à partir de la position4)
6
```

```
>>> fruits.reverse()  # inverser la liste des fruits
>>> fruits
['banane', 'pomme', 'kiwi', 'banane', 'poire', 'pomme', 'orange']
>>> fruits.append('grenade')
>>> fruits
['banane', 'pomme', 'kiwi', 'banane', 'poire', 'pomme','orange',
'grenade']
>>> fruits.sort()  # trier la liste
>>> fruits
['banane', 'banane', 'grenade', 'kiwi', 'orange', 'poire',
'pomme', 'pomme']
```

1.2/ Listes vs Piles

Les méthodes des listes rendent très facile leur utilisation comme des piles, où le dernier élément ajouté est le premier récupéré (« dernier entré, premier sorti », ou LIFO pour « last-in, first-out »). Pour ajouter un élément sur la pile, utilisez la méthode append(). Pour récupérer l'objet au sommet de la pile, utilisez la méthode pop(), sans indicateur de position.

Exemple :

```
>>> pile = [3, 4, 5]
>>> pile.append(6)
>>> pile.append(7)
>>> pile
[3, 4, 5, 6, 7]
>>> pile.pop()
7
>>> pile
[3, 4, 5, 6]
>>> pile.pop()
6
>>> pile.pop()
5
>>> pile
[3, 4]
```

1.3/ Listes vs files

Il est également possible d'utiliser une liste comme une file, où le premier élément ajouté est le premier récupéré (« premier entré, premier sorti », ou FIFO pour « first-in, first-out ») ; toutefois, les listes ne sont pas très efficaces pour ce type de traitement. Alors que les ajouts et

suppressions en fin de liste sont rapides, les opérations d'insertion ou de retrait en début de liste sont lentes (car tous les autres éléments doivent être décalés d'une position).

Pour implémenter une file, utilisez donc la classe collections.deque qui a été conçue pour fournir des opérations d'ajout et de retrait rapides aux deux extrémités.

Exemple :

```
>>> from collections import deque
>>> queue = deque(["Ali", "Karim", "Mounir"])
>>> queue.append("Salim")          # ajouter Salim
>>> queue.append("Rafik")          # ajouter Rafik
>>> queue.popleft()                # defiler le premier element
'Ali'
>>> queue.popleft()                # defiler le deuxième element
'Karim'
>>> queue             # afficher les elements restants de la queue
deque(['Mounir', 'Salim', 'Rafik'])
```

2. Les ensembles

Un ensemble est une collection non ordonnée sans élément dupliqué. Les ensembles supportent également les opérations mathématiques comme l'union, l'intersection et la différence.

Des accolades pour la fonction `set()` peuvent être utilisés pour créer des ensembles. un ensemble vide est noté par `set()`.

```
>>> x =  set()    créer un ensemble vide
>>> print(x)
set()
>>> x = set([0,1,2,3])
>>> print(x)      créer un ensemble non vide
{0,1,2,3}
>>> ens_fruit = {'pomme', 'orange', 'pomme', 'poire', 'orange',
'banane'}
>>> print(ens_fruit)     # les elements dupliqués sont supprimés
{'orange', 'banane', 'poire', 'pomme'}
>>> 'orange' in ens_fruit
True
>>> 'grenade' in ens_fruit
False
>>> a = set('abracadabra')
>>> b = set('alacazam')
```

```
>>> a                              # lettres non répétés dans a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                          # lettres dans a mais non dans b
{'r', 'd', 'b'}
>>> a | b                  # lettres soit dans a ou dans b
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                  # lettres dans a et b à la fois
 {'a', 'c'}
>>> a ^ b                  # lettres dans a ou b mais pas les deux
{'r', 'd', 'b', 'm', 'z', 'l'}
>>> couleur = set()
>>> couleur.add('rouge')
>>> print(couleur)
{'rouge'}
>>> couleur.update(['vert','gris'])
>>> print(couleur)
{'rouge','vert','gris'}
>>> num = set([0,1,2,5])
>>> num.pop()
>>> print(num)
{1,2,5}
```

3. Les tuples

Les tuples correspondent aux listes à la différence qu'ils sont non modifiables. ils utilisent les parenthèses au lieu des crochets :

```
>>> x = (1,2,3)
>>> x
(1, 2, 3)
>>> x[2]
3
>>> x[0:2]
(1, 2)
>>> x[2] = 15
Error
```

L'affectation et l'indiçage fonctionne comme avec les listes, mais si l'on essaie de modifier un des éléments du tuple, Python renvoie un message d'erreur. Si vous voulez ajouter un élément(ou le modifier), vous devez créer un autre tuple :

```
>>> x = (1,2,3)
>>> x + (2,)
(1, 2,3,2)
```

4. Les dictionnaires

À la différence des séquences, qui sont indexées par des nombres, les dictionnaires sont indexés par des *clés*, qui peuvent être de n'importe quel type ; les chaînes de caractères et les nombres peuvent toujours être des clés. Des tuples peuvent être utilisés comme clés s'ils ne contiennent que des chaînes, des nombres ou des tuples ; si un tuple contient un objet mutable, de façon directe ou indirecte, il ne peut pas être utilisé comme une clé. Vous ne pouvez pas utiliser des listes comme clés, car les listes peuvent être modifiées en place en utilisant des affectations par position, par tranches ou via des méthodes comme append().

Le plus simple est de considérer les dictionnaires comme des ensembles non ordonnés de paires *clé: valeur*, les clés devant être uniques (au sein d'un dictionnaire). Une paire d'accolades crée un dictionnaire vide : `{}`. Placer une liste de paires *clé:valeur* séparées par des virgules à l'intérieur des accolades ajoute les valeurs correspondantes au dictionnaire ;

Les principales opérations effectuées sur un dictionnaire consistent à stocker une valeur pour une clé et à extraire la valeur correspondant à une clé. Il est également possible de supprimer une paire clé:valeur avec `del`. Si vous stockez une valeur pour une clé qui est déjà utilisée, l'ancienne valeur associée à cette clé est perdue. Si vous tentez d'extraire une valeur associée à une clé qui n'existe pas, une exception est levée.

Exécuter `list(d.keys())` sur un dictionnaire `d` retourne une liste de toutes les clés utilisées dans le dictionnaire, dans un ordre arbitraire (si vous voulez qu'elles soient triées, utilisez `sorted(d.keys()))`. [2] Pour tester si une clé est dans le dictionnaire, utilisez le mot-clé `in`.

Exemple :

```
>>> tel = {'ali': 4098, 'nabil': 4139}
>>> tel['kamel'] = 4127
>>> tel
{'nabil': 4139, 'kamel': 4127, 'ali': 4098}
>>> tel['ali']
4098
>>> del tel['nabil']
>>> tel['djamel'] = 4127
>>> tel
{'kamel': 4127, 'djamel': 4127, 'ali': 4098}
>>> list(tel.keys())
['kamel', 'djamel', 'ali']
>>> sorted(tel.keys())
['ali', 'djamel', 'kamel']
>>> 'ali' in tel
True
```

```
>>> 'kamel' not in tel
False
```

Le constructeur dict() fabrique un dictionnaire directement à partir d'une liste de paires clé-valeur stockées sous la forme de tuples :

```
>>> dict([('nabil', 4139), ('djamel', 4127), ('ali', 4098)])
{'nabil': 4139, 'ali': 4098, 'djamel': 4127}
```

De plus, il est possible de créer des dictionnaires par compréhension depuis un jeu de clef et valeurs :

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

5. Conditions et boucles

Condition "if "et "if else"

```
a = 10
>>> if a > 5:
...     a = a + 1
...
>>> a
11
```

```
>>> a = 20
>>> if a > 5:
...     a = a + 1
... else:
...     a = a - 1
...
>>> a
21
```

Condition "elif "

Il est possible d'ajouter autant de conditions précises que l'on souhaite en ajoutant le mot clé elif, contraction de "else" et "if", qu'on pourrait traduire par "sinon".

```
>>> a = 5
>>> if a > 5:
...     a = a + 1
... elif a == 5:            % sinon  si a = 5 alors  a =a+1000
...     a = a + 1000
```

```
... else:
...     a = a - 1
...
>>> a
1005
```

Les comparaisons possibles

Il est possible de comparer des éléments:

```
==      égal à
!=      différent de
>       strictement supérieur à
>=      supérieur ou égal à
<       strictement inférieur à
<=      inférieur ou égal à
```

Chainer les comparateurs

Il est également possible de chainer les comparateurs:

```
>>> a, b, c = 1, 10, 100
>>> a < b < c
True
>>> a > b < c
False
```

La boucle while

```
>>> i = 0
>>> while i < 10:
...     print("Poser une question")
...     i = i +1
...
```

La boucle for

```
liste = [1,5,10,15,20,25]
>>> for i in liste:
...     if i > 15:
...             print "On stoppe la boucle"
...             break
...     print i
...

1
5
10
15
On stoppe la boucle
```

6. Fonctions

Fonctions prédéfinis

1- print (« texte », sep= « »)
2- Input( ). L'utilisateur fait entrer des caractères au clavier et termine avec <*Enter*>.

la fonction input() renvoie toujours une chaine de caractères. Si vous souhaitez que l'utilisateur entre une valeur numérique, vous devrez donc convertir la valeur entrée (qui sera donc de toute façon de type string) en une valeur numérique du type qui vous convient, par l'intermédiaire des fonctions intégrées int() (un entier) ou float() (un réel).

Il existe beaucoup de fonctions prédéfinies et qui sont intégrés dans des programmes appelés modules. L'accès à ces fonctions se fait par le biais de l'instruction import.

La définition d'une fonction est la suivante :

```
def nomDeLaFonction(liste de paramètres):
    ...
      bloc d'instructions
    ...
    return …
```

Exemple

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


>>> carre(4)
16
>>> res = carre(3)
>>> Print(res)
9
```

7. Modules

Les modules sont des programmes Python qui contiennent des fonctions que l'on est amené à réutiliser souvent (on les appelle aussi bibliothèques ou libraries). La plupart de ces modules sont déjà installés dans les versions standards de Python.

Exemple

```
>>> import random
>>> random.randint(0,10)
4
```

L'instruction import permet d'accéder à toutes les fonctions du module random ensuite, nous utilisons la fonction (ou méthode) randint(a,b) du module random. Cette fonction renvoie un nombre entier aléatoirement entre a inclus et b inclus(contrairement à range() par exemple). Remarquez la notation objet random.randint() où la fonction randint() peut être considérée comme une méthode de l'objet random. Il existe un autre moyen d'importer une ou des fonctions d'un module :

```
>>> from random import randint
>>> randint(0,10)
7
```

À l'aide du mot-clé from, vous pouvez importer une fonction spécifique d'un module donné. Remarquez que dans ce cas il est inutile de répéter le nom du module, seul le nom de la fonction est requis.

On peut également importer toutes les fonctions d'un module :

```
>>> from random import *

>>> x = [1, 2, 3, 4]
>>> shuffle(x)
>>> x
[2, 3, 1, 4]
>>> shuffle(x)
>>> x
[4, 2, 1, 3]
>>> randint(0,50)
46
>>> uniform(0,2.5)
0.64943174760727951
```

8. Opérations sur les fichiers

Lecture d'un fichier

Créer un fichier dans un éditeur de texte que vous enregistrez dans votre répertoire avec un nom par exemple cours.txt . Ce fichier contient ce qui suit :

Python

Intelligence

Image

Ensuite nous testons cet exemple à l'aide des commandes suivantes :

- fich = open('cours.txt', 'r') le fichier cours est ouvert en mode lecture
- fich.readlines( ) lecture de toutes les lignes contenues dans le fichier
- fich.close( ) fermeture du fichier

  fich.read( ) lit tout le contenu du fichier et renvoie une chaine de caractères
- fich.readline( ) lit une ligne d'un fichier et la renvoie sous forme de chaine de caractères

```
>>> fich = open('cours.txt', 'r')
>>> lignes = fich.readlines()
>>> lignes
['python\n',  'intelligence\n', 'image\n']

>>> for ligne in lignes:    ou bien   for  j  in fich :
...     print ligne
print  j
...
python
intelligence
image

>>> fich.read()
'python\nintelligence\nimage\n'

>>> fich.readline( )
'python'

>>> fich.close()
```

Ecriture dans un fichier

```
>>> liste =[ 'oriente objet', 'theorie des graphes', 'reseaux']
>>> fich = open('cours.txt', 'w')      fichier cours  ouvert en
mode écriture
>>> for jj  in  liste:
…         fich.write(jj)
…
>>> 'oriente objettheorie des graphesreseaux
```