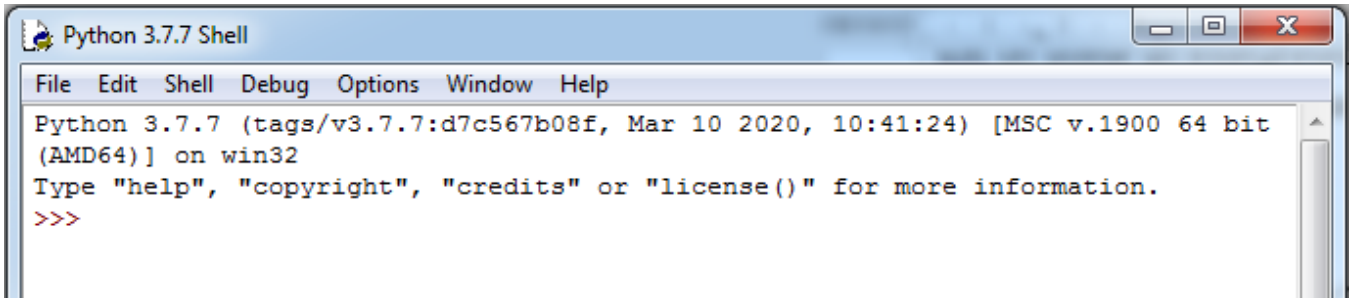
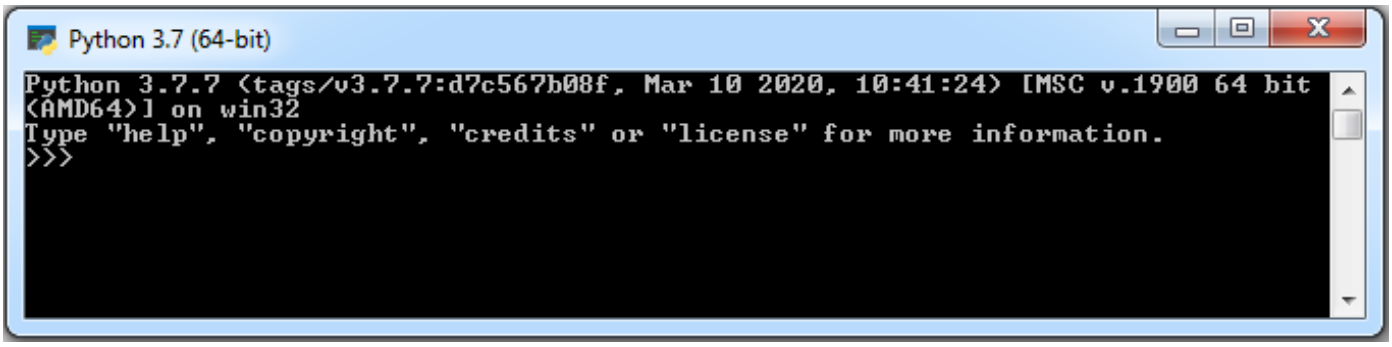
Prise en main en mode interactif

Pour une utilisation en mode interactif, il suffit de lancer l'interpréteur "**IDLE**" qui se trouve dans le dossier d'installation de Python 3.

Une fenêtre "Python Shell" s'ouvre alors. L'invite de commande >>> signifie que Python est prêt à exécuter une commande.



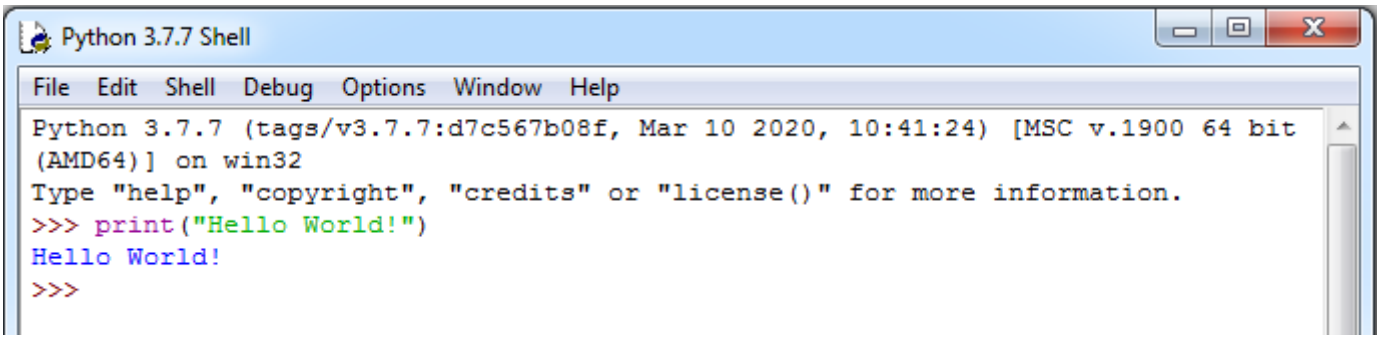
Il est également possible d'utiliser la console Python, également dans le dossier d'installation de Python 3 :



La première commande d'apprentissage de tous les langages de programmation est celle qui affiche la traditionnelle phrase **" Hello World ! "** :

Tapez : **print("Hello World !")**, puis appuyez sur la touche Entrée.

Python exécute cette commande. Le résultat de cette exécution est l'affichage de la chaîne de caractères " Hello World ! ". Une nouvelle invite de commande apparaît alors.



Avec Python, on peut également faire des calculs :

```
================================ RESTART: Shell ================================
>>> 3+2*5
13
>>> 5/2
2.5
>>> 3*(5+1/2)
16.5
>>> 2**5
32
>>> 12%5
2
>>>
```

A noter :

. la virgule des nombres décimaux doit être remplacée par le point.

. 2 exposant 5 s'écrit 2**5

. 12%5 renvoie le reste de la division euclidienne de 12 par 5.

1.1. Variables et affectation

Comme tout langage, Python permet de manipuler des données grâce à un vocabulaire de mots réservées et grâce à des types de données. Il utilise des identifiants pour nommer ses objets.

Un identifiant Python valide est une suite non vide de caractères, de longueur quelconque, formée d'un caractère de début et de zéro ou plusieurs caractères de continuation, sachant que :

- un caractère de début peut être n'importe quelle lettre,
- un caractère de continuation est un caractère de début, un chiffre ou un point.

Attention :

Les identifiants sont sensibles à la casse (majuscules ou minuscules) et ne doivent pas être un mot clé réservé de Python tel que :

and; del; from; None; True; as; elif; global; nonlocal; try; assert; else; if; not; while; break; except; import; or; with; class; False; in; pass; yield; continue; finally; is; raise; def; for; lambda; return.

. Les principaux types de données

Il existe différents types de données : le type entier (int), le type nombre à virgule (float), le type Booléen (bool), le type chaîne de caractères (str), …

- Le type int :

Il représente les nombres entiers. Le type **int** n'est limité en taille que par la mémoire de la machine.
Les entiers sont décimaux par défaut, mais on peut aussi utiliser les bases binaire, octale ou hexadécimale.

On peut effectuer les opérations arithmétiques classiques avec des données de type **int :**

```
20 + 3      # 23
20 - 3      # 17
20 * 3      # 60
```

```
20 ** 3       # 8000
20 / 3        # 6.666666666666667
20 // 3       # 6    (division entière)
20 % 3        # 2     (modulo)
abs(3 - 20)   # 17   (valeur absolue)
```

- Le type float :

Un float est un nombre décimal à virgule noté avec un point décimal ou en notation exponentielle :

```
2.718
.02
3e8
6.023e23
```

Les flottants supportent les mêmes opérations que les entiers.

L'import du module math autorise toutes les opérations mathématiques usuelles :

```
import math
print(math.sin(math.pi/4))          # 0.7071067811865475
print(math.degrees(math.pi))        # 180.0
print(math.factorial(9))            # 362880
print(math.log(1024, 2))            # 10.0
```

- Le type bool :

Une donnée de type bool à deux valeurs possibles : **False** et **True**.

Les opérations logiques et de comparaisons sont évaluées afin de donner des résultats booléens False et True :

- Opérateurs de comparaison : **==**, **!=**, **>**, **>=**, **<** et **<=** :

```
2 > 8            # False
2 <= 8 < 15      # True
```

- Opérateurs logiques: **not**, **or** et **and**.

```
(3 == 3) or (9 > 24)        # True
(9 > 24) and (3 == 3)       # False
```

- **Le type str :**

Le type de données **str** représente une séquence de caractères entre guillemets simple ' ou double " ce qui permet d'inclure une notation dans l'autre :

. guillemets = " L'eau vive "
. apostrophes = ' Forme "avec des apostrophes" '

. Les variables

On utilise les variables pour stocker des données. Une variable est un identifiant associé à une valeur. Informatiquement, c'est une référence d'objet située à une adresse mémoire.

On affecte une variable par une valeur en utilisant le signe = (qui n'a rien à voir avec l'égalité en math !). Dans une affectation, le membre de gauche reçoit le membre de droite ce qui nécessite d'évaluer la valeur correspondant au membre de droite avant de l'affecter au membre de gauche.

Exemple en mode interactif :

```
>>> pi=3.1415
>>> R=3
>>> Adisc=pi*R**2
>>> Adisc
28.273500000000002
```

. L'exécution de la première ligne crée une variable nommée pi contenant la valeur réelle 3.1415.
. L'exécution de la deuxième ligne crée une variable nommée R contenant la valeur entière 3.
. L'exécution de la troisième ligne crée une variable nommée Adisc contenant le résultat du calcul $pi*R^2$.

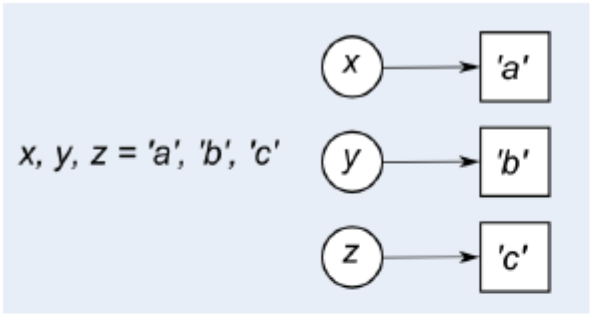
Pour afficher la valeur d'une variable, il suffit de taper son nom puis d'appuyer sur la touche Entrée ou bien taper print(Nom_de_la_variable) :

```
>>> print(Adisc)
28.273500000000002
>>>
```

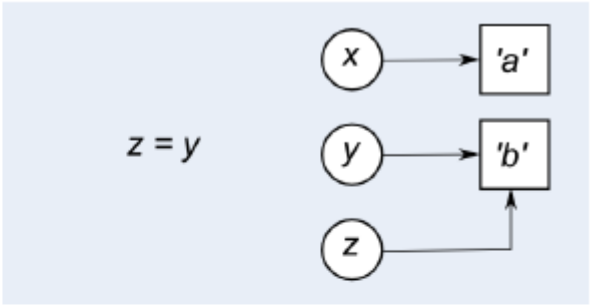
La valeur d'une variable, comme son nom l'indique, peut évoluer au cours du temps (la valeur antérieure est perdue) :

```
>>> a = 2
>>> a = a + 1
>>> a
3
>>> a = a - 1
>>> a
2
```

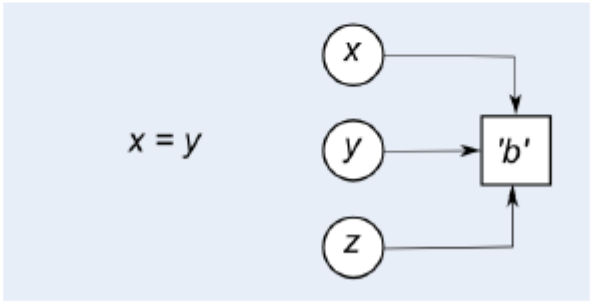
Ceci est résumé dans le schéma suivant, où les cercles représentent les identificateurs (variables) alors que les rectangles représentent les données.
Les affectations relient les identificateurs aux données : si une donnée en mémoire n'est plus reliée, le ramasse-miettes (garbage collector) de Python la supprime automatiquement :



(a) Trois affectations



(b) La donnée 'c' est supprimée



(c) La donnée 'a' est supprimée

Outre l'affectation simple, on peut aussi utiliser les formes suivantes :

```
# affectation simple
v = 4

# affectation augmentée
v += 2 # idem à : v = v + 2 si v est déjà référencé
```

```
# affectation de droite à gauche
c = d = 8 # cibles multiples

# affectations parallèles d'une séquence
e, f = 2.7, 5.1 # tuple
g, h, i = ['G', 'H', 'I'] # liste
x, y = coordonneesSouris() # retour multiple d'une fonction
```

Avec Python, il n'est pas nécessaire de définir préalablement le type de la variable. Le typage se fait automatiquement lors de l'affectation d'une valeur à la variable :

Exemples :

```
>>> a="Hello World !"
>>> b=3
>>> c=2.5
>>> d=[7,3,145]
>>> e=False
>>>
```

- La variable **a** contient une chaîne de caractères, elle sera de type str.

Dès que la valeur d'affection d'une variable est entre guillemets, la variable sera du type « chaîne de caractères » (**str**). Par exemple, si vous saisissez **a="3"** (ou **a='3'**), la variable a est du type chaîne de caractères et la valeur de **a** n'est pas considérée comme un nombre mais comme du texte (Effectuer l'opération a+2 n'aurait aucun sens !) .

- La variable **b** contient un entier, elle sera de type int.

- La variable c contient un nombre à virgule, elle sera de type float.

- La variable d contient une liste, elle sera du type list.

- La variable e contient un booléen, elle sera du type bool (une variable de type bool peut prendre 2 valeurs True ou False).

Pour connaître le type d'une variable il suffit de taper type(nom_de_la_variable) :

```
>>> type(pi)
<class 'float'>
>>>
```

1.2. Les chaines de caractères

Les chaines de caractères sont des données de type **str** représentant une séquence de caractères entre guillemets simple ' ' ou double " ".

Trois syntaxes de chaînes sont disponibles :

```
syntaxe1 = "Première forme sans un retour à la ligne"
syntaxe2 = "Deuxième forme avec retour à la ligne\n "
syntaxe3 = """
           Troisième forme multi-ligne
           Troisième forme multi-ligne
           """
```

Ce qui donne dans la console Python:

```
>>> syntaxe1 = "Première forme sans retour à la ligne"
>>> syntaxe2 = "Deuxième forme avec retour à la ligne\n"
>>> print(syntaxe1,syntaxe2,syntaxe1)
Première forme sans retour à la ligne Deuxième forme avec retour à la ligne
 Première forme sans retour à la ligne
>>>
>>> syntaxe3 = """
               Troisième forme multi-ligne
               Troisième forme multi-ligne
                """
>>> print(syntaxe3)

               Troisième forme multi-ligne
               Troisième forme multi-ligne
```

On peut effectuer des opérations sur les chaines :

. Détermination de la longueur de la chaine à l'aide de la fonction **len()**:

```
>>> s = "abcde"
>>> print(len(s))
5
```

. Concaténation de 2 chaines:

```
>>> s1 = "abc"
>>> s2 = "def"
>>> s3 = s1 + s2
>>> print(s3)
abcdef
```

. Répétition d'une chaine :

```
>>> s4 = "A"
>>> s5 = 3*s4
>>> print(s5)
AAA
```

Les chaines sont des objets auxquels on peut appliquer une méthode en utilisant la "notation pointée" entre la donnée/variable à laquelle on applique la méthode et le nom de la méthode : **chaine.methode()**

. Méthodes de test de l'état d'une chaine ch

Les méthodes couramment utilisées suivantes sont à valeur booléennes, c'est-à-dire qu'elles retournent la valeur True ou False.

. **isupper()** et **islower()** retournent True si **ch** ne contient respectivement que des majuscules/minuscules :

```
>>> ch="abcdef"
>>> print(ch.isupper())
False
>>> print(ch.islower())
True
```

. **istitle()** retourne True si seule la première lettre de chaque mot de ch est en majuscule :

```
>>> ch="Abcdef"
>>> print(ch.istitle())
True
```

. **isalnum()**, **isalpha()**, **isdigit()** et **isspace()** retournent True si ch ne contient respectivement que des caractères alphanumériques, alphabétiques, numériques ou des espaces :

```
>>> ch="abcdef"
>>> print(ch.isalnum())
True
>>> ch="abcdef123"
>>> print(ch.isalpha())
False
>>> ch="abcdef"
>>> print(ch.isalpha())
True
```

```
>>> ch="abcdef123"
>>> print(ch.isdigit())
False
>>> ch="abcdef 123"
>>> print(ch.isspace())
False
>>> ch="   "
>>> print(ch.isspace())
True
```

. Méthodes souvent utilisées retournant une nouvelle chaine

. **lower()**, **upper()**, **capitalize()** et **swapcase()** retournent respectivement une chaine en minuscule, en majuscule, en minuscule commençant par une majuscule, ou en casse inversée :

```
>>> ch = "aBcDEf"
>>> print(ch.lower())
abcdef
>>> print(ch.upper())
ABCDEF
>>> print(ch.capitalize())
Abcdef
>>> print(ch.swapcase())
AbCdeF
```

. **strip('chars')**, **lstrip('chars')** et **rstrip('chars')** suppriment toutes les combinaisons de 'chars' (ou l'espace par défaut) respectivement au début et en fin, au début, ou en fin d'une chaine :

```
>>> ch = "AabcdefA"
>>> print(ch.strip('A'))
abcdef
>>> ch = "AabcdAefg"
>>> print(ch.lstrip('A'))
abcdAefg
>>> ch = "AabcdAefgA"
>>> print(ch.rstrip('A'))
AabcdAefg
```

```
>>> ch=" abcdef "
>>> print(ch)
 abcdef
>>> print(ch.lstrip())
abcdef
```

. **split**('sep', maxsplit) découpe la chaine en maxsplit morceaux (tous par défaut) suivant le séparateur 'sep' (ou l'espace par défaut) :

```
>>> ch="azerty1 azerty2 azerty3"
>>> print(ch.split())
['azerty1', 'azerty2', 'azerty3']
>>> print(ch.split(' ',1))
['azerty1', 'azerty2 azerty3']
```

. **rsplit()** effectue la même chose en commençant par la fin :

```
>>> print(ch.rsplit(' ',1))
['azerty1 azerty2', 'azerty3']
```

. **splitlines()** effectue le même travail mais avec les caractères de fin de ligne :

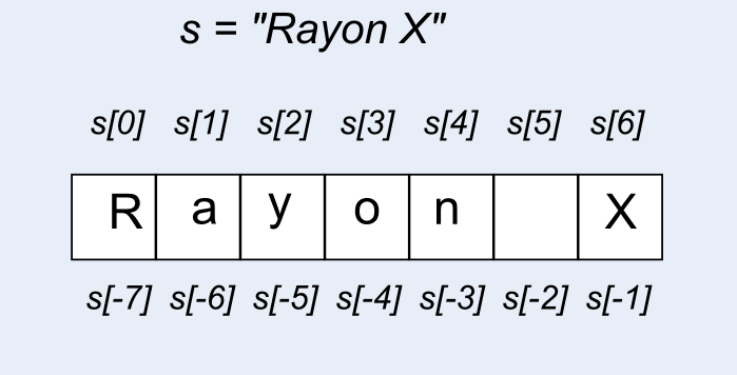
```
>>> ch="azerty1\nazerty2\nazerty3"
>>> print(ch)
azerty1
azerty2
azerty3
>>> print(ch.splitlines())
['azerty1', 'azerty2', 'azerty3']
```

. Indiçage des chaines de caractères

Les chaines de caractères peuvent être indicées avec l'opérateur [ ] dans lequel l'indice, un entier signé qui commence à 0 indique la position d'un caractère :

```
s = "Rayon X"          # len(s) ==> 7
print(s[0])            # R
print(s[2])            # y
print(s[-1])           # X
print(s[-3])           # n
```

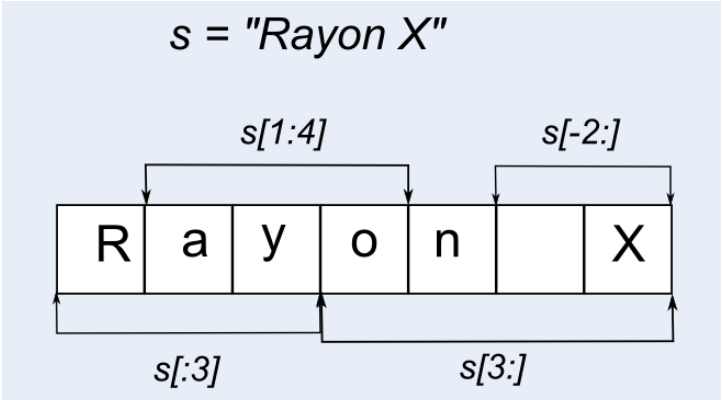
Ci-dessous, un schéma représentatif de l'indiçage de la chaine **s**:



Il est possible d'extraire des sous-chaines de la chaine s :

```
s = "Rayon X"          # len(s) ==> 7
s[1:4]                 # 'ayo' (de l'indice 1 compris à 4 non compris)
s[-2:]                 # ' X' (de l'indice -2 compris à la fin)
s[:3]                  # 'Ray' (du début à 3 non compris)
s[3:]                  # 'on X' (de l'indice 3 compris à la fin)
s[::2]                 # 'RynX' (du début à la fin, de 2 en 2)
```

Opérations d'extraction résumées ci-dessous :



1.3 Les listes

En python, les listes sont des variables qui peuvent contenir n'importe quel type de données.
Elles sont notées sous forme d'éléments entre crochets séparés par des virgules.

```
>>> l=['A',2,3,"azerty"]
>>> print(l)
['A', 2, 3, 'azerty']
```

La numérotation des éléments des listes commence à 0.

```
>>> l=['A',2,3,"azerty"]
>>> print(l[0])
A
>>> print(l[2])
3
```

Les listes correspondent à des objets auxquels, il est possible d'appliquer des méthodes :

. **len()** renvoie le nombre d'éléments de la liste :

```
>>> l=['A',2,3,"azerty"]
>>> print(len(l))
4
```

. **append(e)** ajoute un élément **e** à la liste :

```
>>> l.append(5)
>>> print(l)
['A', 2, 3, 'azerty', 5]
```

. **sort()** trie les éléments de la liste si elle contient des données du même type :

```
>>> l=[48,26,75,14]
>>> l.sort()
>>> print(l)
[14, 26, 48, 75]
```

```
>>> l = ['a','j','b','s','azerty']
>>> l.sort()
>>> print (l)
['a', 'azerty', 'b', 'j', 's']
```

. **remove(e)** retire l'élément e de la liste :

```
>>> l=['A',2,3,"azerty"]
>>> l.remove(2)
>>> print(l)
['A', 3, 'azerty']
```

. **pop()** enlève le dernier élément de la liste :

```
>>> l=['A',2,3,"azerty"]
>>> dernier_element=l.pop()
>>> print(l)
['A', 2, 3]
>>> print(dernier_element)
azerty
```

. **pop(i)** enlève l'élément d'indice **i** de la liste :

```
>>> l=['A',2,3,"azerty"]
>>> element_1=l.pop(1)
>>> print(l)
['A', 3, 'azerty']
>>> print(element_1)
2
```

. **index(e)** retourne la position de l'élément **e** de la liste **:**

```
>>> l=['A',2,3,"azerty"]
>>> print(l.index(2))
1
```

. **reverse()** inverse l'ordre des éléments de la liste :

```
>>> l=['A',2,3,"azerty"]
>>> l.reverse()
>>> print(l)
['azerty', 3, 2, 'A']
```

. **count(e)** compte le nombre d'occurrence de l'élément e dans la liste :

```
>>> l=[1,5,8,5,6,4]
>>> print(l.count(2))
0
>>> print(l.count(5))
2
```

. **extend()** concatène deux listes :

```
>>> l1=['A',2,3,"azerty"]
>>> l2=['B',4,5]
>>> l1.extend(l2)
>>> print(l1)
['A', 2, 3, 'azerty', 'B', 4, 5]
```

Remarques :

. Une liste **l** vide s'écrit : **l = []**

. La fonction **del** permet de supprimer un élément d'index **i** d'une liste :

```
>>> l=['A',2,3,"azerty"]
>>> del l[0]
>>> print(l)
[2, 3, 'azerty']
```

. Les expressions d'indiçage des chaines de caractères s'appliquent aussi aux listes :

```
>>> l=['A',2,3,"azerty"]
>>> print(l[1:2])
[2]
>>> print(l[1:3])
[2, 3]
>>> print(l[-1])
azerty
>>> print(l[:-2])
['A', 2]
>>> print(l[-2:])
[3, 'azerty']
```

. La méthode **split()** permet de transformer une chaine de caractère en liste :

```
>>> s="je veux couper la chaine"
>>> l=s.split(' ')
>>> print(l)
['je', 'veux', 'couper', 'la', 'chaine']
```

. La méthode **join()** permet de transformer une liste de chaine en une chaine de caractère :

```
>>> s=' '.join(l)
>>> print(s)
je veux couper la chaine
```

. En plus de la méthode **count()**, on peut également savoir si un élément est dans une liste, en utilisant le mot clé **in** de cette manière:

```
>>> liste = [1,2,3,5,10]
>>> 3 in liste
True
>>> 6 in liste
False
```

. A la place de la méthode **extend()** , on peut additionner deux listes pour les combiner ensemble en utilisant l'opérateur **+** :

```
>>> l1=['A',2,3,"azerty"]
>>> l2=['B',4,5]
>>> l3=l1+l2
>>> print(l3)
['A', 2, 3, 'azerty', 'B', 4, 5]
```

. Il est également possible de multiplier des listes :

```
>>> l=['a','b']
>>> l=l*5
>>> print(l)
['a', 'b', 'a', 'b', 'a', 'b', 'a', 'b', 'a', 'b']
```

ce qui est pratique pour initialiser une liste :

```
>>> l=[0]*5
>>> print(l)
[0, 0, 0, 0, 0]
```

. La fonction **range()** génère une liste composée d'une simple suite arithmétique :

```
>>> liste = range(5)
>>> print(liste)
range(0, 5)
```

- La fonction **list()** crée une liste (à partir d'une chaine, d'un tuple ou d'une liste) :

```
>>> list(liste)
[0, 1, 2, 3, 4]
>>> l=['A',2,3,"azerty"]
>>> list(l)
['A', 2, 3, 'azerty']
```

- On peut préciser l'entier de départ (**range(entier de départ inclus, entier de fin exclu)**) :

```
>>> liste = range(3,10)
>>> list(liste)
[3, 4, 5, 6, 7, 8, 9]
```

- Et l'incrément (**range(entier de départ inclus, entier de fin exclu, incrément)**) :

```
>>> liste = range(2,10,2)
>>> list(liste)
[2, 4, 6, 8]
```

. Pour afficher les éléments d'une liste on peut aussi utiliser une boucle **For** :

```
>>> liste = range(5)
>>> print(liste)
range(0, 5)
>>> for elt in liste: print(elt)

0
1
2
3
4
```

. La fonction **enumerate**() permet en plus de récupérer l'index de l'élément :

```
>>> liste=['A',2,3,"azerty"]
>>> for elt in enumerate(liste): print(elt)

(0, 'A')
(1, 2)
(2, 3)
(3, 'azerty')
```

Les valeurs retournées par la boucle sont des tuples.

1.4 Les tuples

Les tuples sont des listes qui ne peuvent pas être modifiées. Ils sont notés sous forme d'éléments entre parenthèses séparés par des virgules.

```
>>> t = ("un", 2, "trois")
>>> print(t)
('un', 2, 'trois')
```

Les parenthèses ne sont pas obligatoires :

```
>>> t = 1,2,3
>>> type(t)
<class 'tuple'>
```

Le tuple étant une sorte de liste, on peut donc utiliser la même syntaxe pour lire les données du tuple.

```
('un', 2, 'trois')
>>> print(t[1])
2
```

Mais si on essaie de changer la valeur d'un index, l'interpréteur affiche un message d'erreur :

```
>>> t[1]='deux'
Traceback (most recent call last):
  File "<pyshell#123>", line 1, in <module>
    t[1]='deux'
TypeError: 'tuple' object does not support item assignment
```

Cependant, le tuple permet une affectation multiple :

```
>>> t = (1,2)
>>> var1,var2 = t
>>> print(var1)
1
>>> print(var2)
2
```

Ou

```
>>> var3,var4 = 3,4
>>> print (var3, var4)
3 4
```

Il sera donc principalement utilisé pour définir les constantes des programmes.

1.5 Les dictionnaires

Comme les listes, les dictionnaires permettent de stocker des données mais au lieu d'utiliser des index pour les repérer, on utilise des clés alphanumériques.

Chaque élément d'un dictionnaire est composé de 2 parties, on parle de paires "clé/valeur".

- Pour ajouter des données à un dictionnaire il faut donc indiquer une clé ainsi qu'une valeur :

. Création d'un dictionnaire :

dictionnaire = {clé1:valeur1, clé2:valeur2,…}

dictionnaire = {} (dictionnaire vide)

ou

dictionnaire = dict([(clé1,valeur1), (clé2:valeur2),…]) (liste de tuples)

dictionnaire = dict() (dictionnaire vide)

. Ajout d'une donnée :

```
>>> inventaire = {}
>>> inventaire["bécher"]= 10
>>> print(inventaire)
{'bécher': 10}
>>> inventaire["éprouvette"]= 20
>>> print(inventaire)
{'bécher': 10, 'éprouvette': 20}
```

- La méthode **get ()** permet de récupérer une valeur du dictionnaire :

```
>>> print(inventaire.get('bécher'))
10
```

- pour effacer une entrée (clé/valeur), on utilise la fonction **del** :

```
>>> del inventaire["bécher"]
>>> inventaire
{'éprouvette': 20}
```

- A l'aide d'une boucle **for** et de la méthode **keys()**, on peut récupérer les clés d'un dictionnaire :

```
>>> inventaire={"bécher":10,"éprouvette":20}
>>> for cle in inventaire.keys(): print(cle)

bécher
éprouvette
```

- et avec la méthode **values()**, on récupère les valeurs :

```
>>> for valeur in inventaire.values(): print(valeur)

10
20
```

- et pour récupérer les clés et les valeurs en même temps, on utilise la méthode **items()** qui retourne un tuple :

```
>>> inventaire=dict([("bécher",10),("éprouvette",20),("erlenmeyer",15)])
>>> for cle,valeur in inventaire.items():
        print (cle, valeur)


bécher 10
éprouvette 20
erlenmeyer 15
```