# Analyse et Programmation avec Python

Janvier 2020

Enseignant : Jaafar Chaaouri

Email: Jaafar.chaaouri@fsm.rnu.tn

---

## Le langage Python [1/2]

- Variables
- Types de données
  - Types de base
    - None
    - Booléens
    - Numériques (entiers, flottants et complexes)
  - Séquences
    - Chaines de caractères
    - listes
    - tuples
  - Conteneurs
    - Dictionnaires
    - Ensembles
- Fichiers

## Langage python et sa syntaxe

### Variables et assignation

Pour accéder à une donnée, on lui donne un nom que l'on appelle variable. Cette opération s'appelle *assignation* (ou *affectation*) et s
l'opérateur `=`

```
variable_x = donnee_x
```

signifie qu'on affecte la `donnee_x` à la `variable_x`.

Exécutez la cellule suivante pour définir trois variables nommées : `age`, `prenom` et `taille`.

```
# Par exemple: donnons la valeur 23 à la variable 'age'
age = 23
# Les variables peuvent se référer à divers types de données: des chaînes de caractères...
prenom = 'Chahine'
# Des nombres réels, etc...
taille = 1.83
```

### Le mécanisme d'assignation en détail

Lorsqu'on fait l'assignation `x = donnee_x`, Python :

1. crée et mémorise le nom de variable `x`
2. attribue un type à cette variable en fonction de la donnée affectée `donnee_x`
3. crée et mémorise une valeur particulière pour `donnee_x`
4. établit une référence entre le nom de la variable `x` et l'emplacement mémoire de `donnee_x`. Cette référence est mémorisée dans l'espace de n

## Accéder à la valeur d'une variable

Pour se servir de la donnée référencée par une variable, il suffit d'utiliser le nom cette variable.

La fonction `print()` affiche à l'écran ce qui lui est passé en paramètre. On peut lui donner plusieurs paramètres séparés par des vi `print` les affichera tous, séparés par un espace.

```
print('Chahine', 23, 1.83)
print(prenom, 'a', age, 'ans, et mesure', taille, 'mètre')
```

Les variables peuvent changer et ainsi se référer à une autre donnée.

```
age = 23
print(age)
age = 24
print(age)
```

## L'assignation du point de vue de l'objet

En python, toute donnée est vue comme un objet. Assigner une valeur à une variable revient à nommer un objet, c'est-à-dire référenc par un nom.

Exemple avec l'entier `7` :

- pour Python, `7` est l'objet de type entier ( `int` ) dont la donnée est `7`
- lorsqu'on appelle la donnée `7` dans un programme, Python créé l'objet correspondant et lui affecte un identifiant unique dans nom courant.
- cet identifiant est renvoyé par la fonction intrinsèque `id()` :

```
print(id(7))
```

```
 x = 7
```

signifie qu'on nomme l'objet `7` avec la variable `x` . Autrement dit, `x` devient une référence vers cet objet.

```
print(id(7))
x = 7
print(id(x))
y = x
print(id(y))  # y référence le même objet que x
```

## Le type

Les variables n'ont pas de type propre : c'est la donnée qui est typée, pas la variable qui la référence. Une variable peut donc faire ré une donnée d'un autre type après une nouvelle assignation.

La fonction `type()` retourne le type effectif de la donnée passée en paramètre.

```
# Des nombres
age = 23
print(type(age))
```

```
# Les variables peuvent aussi changer de type : chaîne de caractères
age = 'vingt quatre ans'
print(type(age))
```

```
# Sans limites...
age = 24.5
print(type(age))
```

```
# Attention aux pièges...
age = '25'
print(type(age))
```

Une variable peut être initialisée avec des valeurs constantes, comme vu précédement, mais aussi à partir d'autres variables ou des retour de fonctions, etc.

```
a = 1
b = a
c = a * 2
d = max(a, 2, 3, c)
print(a, b, c, d)
```

Les variables, une fois définies dans une cellule **exécutée**, continuent d'exister dans les suivantes.

> **Note :** En cas de redémarrage du notebook, toutes les variables existantes sont détruites, il faut donc ré-exécuter les cellules q les définissent si l'on veut de nouveau pouvoir les utiliser.

```
abcd = 1234
```

Ici, la variable nommée `abcd` survit, d'une cellule à la suivante...

```
print(abcd)
```

Si on veut faire disparaître une variable, on peut utiliser la fonction `del()`.

```
# Cette cellule génère une erreur
a = 2
print(a)
del(a)
print(a)
```

**Note :** `del` est aussi utilisé pour enlever des éléments dans des conteneurs modifiables (listes, dictionnaires). Nous aborderons ce tard.

## Types de données

## Types de base

### None

- Il existe dans python un type de données particulier : `None`.
- `None` représente un objet sans valeur. On s'en sert comme valeur de retour en cas d'erreur ou pour représenter un cas particul
- `None` est équivalent à `NULL` en Java et en C.

```
a = None
print(a)
print(type(a))
```

### Booléens

Les booléens ne peuvent avoir que deux valeurs :

```
True, False
```

On peut utiliser la fonction `bool()` pour construire un booléen.

Dans un contexte booléen, toutes ces valeurs sont équivalentes à `False` :

- `None`
- le zero des types numériques, par exemple : `0, 0.0, 0j`.
- les séquences vides, par exemple : `'', (), []`.
- les dictionnaires vides, par exemple, `{}`.

Tout le reste est équivalent a `True` :

- une valeur numérique différente de zéro
- une séquence non vide
- un dictionnaire non vide

## Exemple d'utilisation d'un contexte booléen

```
Am_I_OK = True
if Am_I_OK:
    print('OK')
else:
    print('KO')
print(Am_I_OK)
print(type(Am_I_OK))
```

## Numériques

### Entiers

La taille est illimitée.

```
# Il n'y a plus de distinction entre les entiers "courts" et les entiers "longs"
entier = 4
print(entier)
print(type(entier))
# Ce nombre nécéssite 131 bits
entier = 14159265358979323846264338327950288419 71
print(entier)
print(type(entier))
```

On peut utiliser la fonction interne `int()` pour créer des nombres entiers. `int()` peut créer des entiers a partir de leur représentati
forme de chaîne de caractères. On peut aussi spécifier la base.

```
entier = int(12)
print(entier)
print(type(entier))
entier = int('13')
print(entier)
print(type(entier))
entier = int('0xFF', 16)
print(entier)
print(type(entier))
```

## Flottants (réels 64 bits)

Pour créer des nombres réels (en virgule flottante), on peut utiliser la fonction interne `float()`.

La précision est limitée à la 16ème décimale.

```
pi_approx = 3.1415926535897932
print(pi_approx)
print(type(pi_approx))
print('{:.16f}'.format(pi_approx))
tropgros = float('Infinity')
print(type(tropgros))
print(tropgros)
```

## Flottants

Attention ! Python autorise un affichage plus long que la précision des flottants mais tous les chiffres après le 16ème chiffre signifi
faux :

```
# On demande 20 chiffres après la virgule, alors que 16 seulement sont exacts
print('{:.20f}'.format(pi_approx))
```

Attention à ne pas effectuer des opérations interdites...

```
# Cette cellule génère une erreur
print(3.14 / 0)
```

```
# Cette cellule génère une erreur
print(34 % 0)
```

```
# Cette cellule génère une erreur
print(27 // 0.0)
```

```
# Cette cellule génère une erreur
print(1 + None)
```

### Complexes

Les nombres complexes, à deux dimensions, peuvent être créés en utilisant le suffixe `j` à la fin d'un entier ou réel ou en utilisant la f
interne `complex()`

```
complexe = 1 + 2j
print('représentation :', complexe)
print(type(complexe))
c = .3j
print(c)
```

## Séquences

Les séquences sont des conteneurs d'objets où les objets référencés sont ordonnés.

Python supporte nativement trois types de séquences :

- les chaînes de caractères
- les listes
- les tuples

### Chaînes de caractères

Pour le traitement de données textuelles, python utilise les chaînes de caractères.

Pour délimiter le texte on utilise des guillemets `"` (double-quote) ou des apostrophes `'` (single-quote).

```
mois1 = 'janvier'
mois2 = "février"
print(mois1, mois2)
```

Les deux formes sont très utiles car elles permettent d'utiliser des guillemets ou des apostrophes dans des chaîne de caractères de
simple.

```
arbre = "l'olivier"
print(arbre)
record_100m = 'neuf secondes et 58 dixièmes (9"58)'
print(record_100m)
```

Sinon, pour avoir une apostrophe ou un guillemet dans une chaîne de caractères, il faut le faire précéder d'un `\` (backslash). Ce qui
beaucoup moins lisible, mais parfois obligatoire (par exemple une chaîne avec à la fois des guillemets et des apostrophes).

```
arbre = 'l\'alisier'
pates = '8\''
record = "9\"58"
print(arbre, pates, record)
duel = 'guillemet: " et apostrophes: \' peuvent être utilisés dans une même chaîne...'
print(duel)
multi = '''Dans une "triplequoted" (voire plus loin), on peut (presque) tout utiliser: `"“ ”,'"«»'"' et ça ne pose pas
print(multi)
```

### Caractères spéciaux

Il est possible de définir des chaînes de caractères qui contiennent des caractères spéciaux. Ils sont introduits par des séquences d
caractères dont le premier est un \ (backslash). On l'appelle le caractère d'échappement.

- retour à la ligne : `\n`
- tabulation : `\t`
- backslash : `\\`
- un caractère unicode avec son code : `\uXXXX` (où les `XXXX` sont le code hexadécimal représentant ce caractère)

Plus d'information dans la documentation officielle

```
print("Une belle présentation, c'est:")
print('\t- bien ordonné')
print('\t- aligné')
print("\t- même s'il y en a plein\nEt c'est plus joli.")
```

## Chaînes multilignes

Pour écrire plusieurs lignes d'une façon plus lisible, il existe les chaînes multilignes :

```
# L'équivalent est :
print("""\
Une belle présentation, c'est:
\t- bien ordonné
\t- aligné
\t- même s'il y en a plein
Et c'est plus joli.""")
```

Les deux formes de délimiteurs sont aussi utilisables : guillemets triples ou apostrophes triples.

```
multi1 = '''m
a
r
s est un mois "multiligne"'''
print(multi1, '\n')

multi2 = """a
v
r
i
l l'est aussi"""
print(multi2)
```

## Unicode

En python 3 , toutes les chaînes de caractères sont unicode, ce qui permet d'utiliser des alphabets différents, des caractères accent pictogrammes, etc.

Exemples de caractères spéciaux :

```
unicode_str = "Les échecs (♔♕♖♗♘♙), c'est \u263A"
print(unicode_str)

Arabic_str= 'با يثون'
print(Arabic_str)
```

Pour une liste de caractères unicode, voir ici.

## Chaînes spéciales

Il existe d'autres manières plus spécialisées de définir des chaînes de caractères.

Premièrement, des chaînes dans lesquelles les séquences d'échappement ne sont pas remplacées (raw strings = chaînes brutes) : `r'''...'''`, etc.

```
normal_str = "chaîne normale: \n'est pas un retour à la ligne, \t pas une tabulation"
print(normal_str)

raw_str = r"chaîne RAW: \n'est pas un retour à la ligne, \t pas une tabulation"
print(raw_str)
```

Plusieurs chaînes de caractères contigües sont rassemblées (concaténées)

```
b = 'a' 'z' 'e' 'r' 't' 'y'
print(b)
```

On peut utiliser la fonction `str()` pour créer une chaîne de caractère à partir d'autres objets.

```
a = 23
```

```
ch_a = str(a)
print(type(a), a)
print(type(ch_a), repr(ch_a))
a = 3.14
ch_a = str(a)
print(type(a), a)
print(type(ch_a), repr(ch_a))
```

On ne peut pas mélanger les guillemets et les apostrophes pour délimiter une chaîne de caractères.

```
# Cette cellule génère une erreur
a = "azerty'
```

```
# Cette cellule génère une erreur
a = 'azerty"
```

> **Exercice** : corrigez les deux cellules ci dessus.

## Le formatage de chaîne de caractère avec `format()`

On peut formater du texte, c'est à dire utiliser une chaîne de caractères qui va servir de modèle pour en fabriquer d'autres. Historiqu existe plusieurs méthodes mais nous ne voyons ici qu'une utilisation basique de la méthode `format()`.

`format()` remplace les occurences de `'{}'` par des valeurs qu'on lui spécifie en paramètre. Le type des valeurs passées n'est pas une représentation sous forme de chaîne de caractère sera automatiquement créée, avec la fonction `str()`.

```
'Bonjour {} !'.format('le monde')
```

```
variable_1 = 27
variable_2 = 'vingt huit'
ch_modele = 'Une chaine qui contient un {} ainsi que {} et là {}'
ch_modele.format('toto', variable_1, variable_2)
```

## Les spécifications de format

Des indicateurs peuvent être fournis à la méthode `format()` pour spécifier le type de donnée à intégrer et la manière de formater s représentation.

- `:d` pour des entiers
- `:s` pour des chaînes de caractères
- `:f` pour des nombres flottants
- `:x` pour un nombre entier affiché en base hexadécimale
- `:o` pour un nombre entier affiché en base octale
- `:e` pour un nombre affiché en notation exponentielle

```
import math
a = 27
print("""un entier : {:d},
une chaîne : {:s},
un flottant avec une précision spécifiée : {:.02f}""".format(a, 'Arte', math.pi))
print('Des hexadécimaux: {:x} {:x} {:x} {:x}'.format(254, 255, 256, 257))
print('Des octaux: {:o} {:o} {:o} {:o}'.format(254, 255, 256, 257))
print('Des nombres en notation exponentielle {:e}'.format(2**64))
```

**Attention :** Si le type de la donnée passée ne correspond pas à la séquence de formatage, python va remonter une erreur.

```
# Cette cellule génère une erreur
variable_3 = 'une chaine de caracteres'
# Exemple d'erreur de type : on fournit une chaîne de caractères
# alors que la méthode attend un entier
print('on veut un entier : {:d}'.format(variable_3))
```

## Listes

- Une liste est un objet pouvant contenir d'autres objets
- Ces objets, appelés éléments, sont ordonnés de façon séquentielle, les uns à la suite des autres

- C'est un conteneur dynamique dont le nombre et l'ordre des éléments peuvent varier

On crée un liste en délimitant par des crochets `[]` les éléments qui la composent :

```
L = ['Janvier', 'Février', 'Mars', 'Mai', 'Juin']
print(L, 'est de type', type(L))
```

Une liste peut contenir n'importe quel type d'objets.

```
L0 = [1, 2]
L2 = [None, True, False, 0, 1, 2**64, 3.14, '', 0+1j, 'abc']
L1 = []
L3 = [[1, 'azerty'], L0]
print(L0, L1)
print(L2)
print(L3)
```

On peut utiliser la fonction `list()` pour créer une liste a partir d'autres séquences ou objets.

```
a = list()
b = list('chaine')
print(a)
print(b)
```

On accède aux éléments d'une liste grâce à un indice. Le premier élément est **indicé 0.**

```
print(L[0])
print(L[4])
```

Un dépassement d'indice produit une erreur :

```
# Cette cellule génère une erreur
print(L[10])
```

Les listes sont dites *muables* : on peut modifier la séquence de ses éléments.

Je remplace le deuxième élément :

```
L[1] = 'tomatoes'
print(L)
L[3] = 9
print(L)
```

### Méthodes associées aux listes

Méthodes ne modifiant pas la liste

- La longueur d'une liste est donnée par fonction `len()`
- `L.index(elem)` renvoie l'indice de l'élément `elem` (le 1er rencontré)

Méthodes modifiant la liste

- `L.append()` ajoute un élément à la fin
- `L.pop()` retourne le dernier élément et le retire de la liste
- `L.sort()` trie
- `L.reverse()` inverse l'ordre

Plus d'infos dans la doc officielle.

## Exemples

```
L= ['Abricot', 'Amande', 'Ananas', 'Banane', 'Cerise']

print('len() renvoie:', len(L))
L.append('Citron') # Ne renvoie pas de valeur
print('Après append():', L)
```

```
print('pop() renvoie:', L.pop())
print('Après pop():', L)


L.reverse() # Ne renvoie pas de valeur
print('Après reverse():', L)


print('index() renvoie:', L.index('Amande'))


L.remove('Banane') # Ne renvoie pas de valeur
print('Après remove:', L)
```

> Dans les exemples précédents, remarquez la syntaxe qui permet d'appliquer une méthode à un objet :
>
> ```
> objet.methode()
> ```
>
> Ici, `.methode()` est une fonction propre au type de `objet`. Si `.methode()` existe pour un autre type, elle n'a pas forcément le même comportement. Nous verrons plus en détail cette mécanique dans la section dédiée à la programmation objet.

Pour obtenir la liste des méthodes associées aux listes, on peut utiliser la fonction interne `help()` :

```
help(list) # ou aussi help([])
```

On peut créer facilement des listes répétitives grâce a l'opération de multiplication.

```
a = ['a', 1] * 5
print(a)
```

Mais on ne peut pas 'diviser' une liste.

```
# Cette cellule génère une erreur
a = [1, 2, 3, 4]
print(a / 2)
```

Vous trouverez la documentation complète sur les listes ici.

## Tuples

Les Tuples (ou n-uplets en Français) sont des séquences *immuables* : on ne peut pas les modifier après leur création.

On les initialise ainsi :

```
T = ('a', 'b', 'c')
print(T, 'est de type', type(T))
T = 'a', 'b', 'c'  # une autre façon, en ommettant les parenthèses
print(T, 'est de type', type(T))
T = tuple(['a', 'b', 'c'])  # à partir d'une liste
print(T, 'est de type', type(T))
T = ('a')  # ceci n'est pas un tuple
print(T, 'est de type', type(T))
T = ('a',)  # Syntaxe pour initialiser un tuple contenant un seul élément
print(T, 'est de type', type(T))
T = 'a',  # Syntaxe alternative pour initialiser un tuple contenant un seul élément. Préférez celle avec parenthèses.
print(T, 'est de type', type(T))
```

Une fois créée, cette séquence ne peut être modifiée.

```
T = ('a', 'b', 'c')
print(T[1]) # On peut utiliser un élément


# Cette cellule génère une erreur
T[1] = 'z'  # mais pas le modifier
```

Intérêt des tuples par rapport aux listes :

- plus rapide à parcourir que les listes
- immuables donc "protégés"
- peuvent être utilisés comme clé de dictionnaires (cf. plus loin)

On peut créer des tuples a partir d'autres séquences ou objets grâce à la fonction `tuple()`.

```
a = [1, 2, 3, 'toto']
print(type(a), a)
b = tuple(a)
print(type(b), b)
a = 'azerty'
print(type(a), a)
b = tuple(a)
print(type(b), b)
```

## Manipulation des tuples

Construire d'autres tuples par concaténation et multiplication

```
T1 = 'a', 'b', 'c'
print('T1 =', T1)
T2 = 'd', 'e'
print('T2 =', T2)
print('T1 + T2 =', T1 + T2)
print('T2 * 3 =', T2 * 3)
```

## Types muables et types immuables

Avant d'aller plus loin dans la revue des types, il est important de comprendre le mécanisme d'affectation en fonction du caractère r
immuable de l'objet.

## Cas d'un objet muable

Deux noms de variables différents peuvent référencer le même objet. Si cet objet est muable, les modifications faites par l'interméd
des variables sont visibles par toutes les autres.

```
a = ['Cerise', 'Citron']  # on initialise la variable a
b = a                # b référence le même objet que a
```

`a` et `b` possèdent la même référence :

```
print(id(a))
print(id(b))
```

`a` et `b` contiennent la même donnée :

```
print(a)
print(b)
```

Si on modifie la donnée de `a`, la donnée de `b` est aussi modifiée !

```
a.append('Amande')
print(a)
print(b)
```

## Cas d'un objet immuable

```
t = 'banane', 'orange'
u = t
print(id(t))
print(id(u))
```

`t` et `u` sont deux variables qui référencent le même objet `tuple` donc leur donnée ne peut être modifiée. Tout ce qu'on peut faire, c
affecter une nouvelle valeur :

```
t = 'Amande', 'citron'
```

Dans ce cas, `t` référence un nouvel objet alors que `u` référence toujours l'objet initial :

```
print(id(t))
print(id(u))
```

Et bien sûr leurs données sont différentes :

```
print(t)
print(u)
```

## Un peu plus loin...

Bien que sa séquence soit immuable, si un `tuple` est constitué d'éléments muables, alors ces éléments-là peuvent être modifiés.

Illustration avec un `tuple` dont un des éléments est une `list` :

```
T = ('a', ['b', 'c']) # le deuxième élément est une liste donc il est mutable
print('T =', T)
L = T[1]
print('L =', L)
L[0] = 'e'
print('L =', L)
print('T =', T)
```

```
# Ici on fait exactement la même chose...
T = ('a', ['b', 'c'])
print('T =', T)
T[1][0] = 'z'
print('T après =', T)
```

```
# Cette cellule génère une erreur
T[0] = 'A'  # Ici, on essaye de modifier le tuple lui même...
```

## Le slicing de séquences en Python

- Cela consiste à extraire une sous-séquence à partir d'une séquence.
- Le slicing fonctionne de manière similaire aux intervalles mathématiques : `[début:fin[`
- La borne de fin ne fait pas partie de l'intervalle sélectionné.
- La syntaxe générale est `L[i:j:k]`, où :
  - `i` = indice de début
  - `j` = indice de fin, le premier élément qui n'est pas sélectionné
  - `k` = le "pas" ou intervalle (s'il est omis alors il vaut `1`)
- La sous-liste sera donc composée de tous les éléments de l'indice `i` jusqu'à l'indice `j-1`, par pas de `k`.
- La sous-liste est un nouvel objet.

Dans le sens normal (le pas `k` est positif)

- Si `i` est omis alors il vaut `0`
- Si `j` est omis alors il vaut `len(L)`

Dans le sens inverse (le pas `k` est négatif)

- Si `i` est omis alors il vaut `-1`
- Si `j` est omis alors il vaut `-len(L)-1`

Illustrons ça en créant une liste à partir d'une chaîne de caractères.

La fonction `split()` découpe une chaîne de caractères en morceaux, par défaut en 'mots'.

```
L = 'Dans le Python tout est bon'.split()
print(L)
```

Pour commencer, on extrait de la liste `L` un nouvel objet liste qui contient tous les éléments de `L` <=> copie de liste

```
print(L[0:len(L):1]) # Cette notation est inutilement lourde car :
print(L[::])         # i = 0, j=len(L) et k=1 donc i, j et k peuvent être omis
print(L[:])          # on peut même ommettre le 2ème ":"
```

On extrait une sous-liste qui ne contient que les 3 premiers éléments :

```
print(L[0:3:1])  # Notation complète
print(L[:3:1])  # Le premier indice vaut i=0, donc on peut l'omettre
print(L[:3])  # Le pas de slicing vaut 1, donc on peut l'omettre, ainsi que le ":"
```

J'extrais une sous-liste qui exclut les trois premiers éléments :

```
print(L[3:len(L):1])  # Cette notation est inutilement lourde car :
print(L[3:])          # j et k peuvent être omis, ainsi que le ":"
```

Les indices peuvent être négatifs, ce qui permet traiter les derniers éléments :

```
# Je veux exclure le dernier élément :
print(L[0:-1:1])  # Notation complète
print(L[:-1:1])  # Le premier indice vaut i=0, donc on peut l'omettre
print(L[:-1])  # Le pas de slicing vaut 1, donc on peut l'omettre
```

On ne garde que les deux derniers éléments

```
print(L[-2:])
```

## Note

`L[1]` n'est pas équivalent à `L[1:2]`, ni à `L[1:]`, ni à `L[:1]`.

Illustration :

```
a = L[1]
print(type(a), a) # Je récupère le deuxième élément de la liste
a = L[1:2]
print(type(a), a) # Je récupère une liste composée du seul élément L[1]
a = L[1:]
print(type(a), a) # Je récupère une liste
a = L[:1]
print(type(a), a) # Je récupère une liste
```

> **Exercice :** Retourner une liste composée des éléments de `L` *en ordre inverse* **avec une opération de slicing**. Toute utilisation de `[].reverse()` ou `reversed()` est interdite.

```
L = 'Dans le Python tout est bon'.split()
print(L)
# <- votre code ici
```

```
# Solution

L = "Dans le Python, tout est bon.".split()

# La solution simple et élégante :
print(L[::-1])

# Explications:
# Pas de bornes => valeurs par défaut => on prend tout
# Le pas est de -1 => à l'envers

# La version explicite :
print(L[-1:-len(L)-1:-1])

# Explications :
# On commence à la dernière place
# On s'arrête à la première, exprimée en indices négatifs
# Le pas est de -1 => à l'envers

# Ne pas oublier que les indices vont:
# dans le sens normal : de 0 à 5
# dans le sens contraire : de -1 à -6

# Ne pas oublier que les bornes d'un slice qui prend tout les éléments vont:
# dans le sens normal : de 0 à 6
# dans le sens contraire : de -1 à -7
```

```
# La solution "optimale" : elle utilise un iterateur et donc
# ne consomme aucune ressource tant qu'on ne l'utilise pas
print(list(reversed(L)))

# Pour le fun, voici une version fonctionelle récursive
def rev(alist):
    if not alist:
        return []
    return alist[-1:] + rev(alist[:-1])

print(rev(L))
```

Le slicing peut être utilisé pour **modifier** une séquence **muable** grâce à l'opération d'assignation.

```
L = 'Dans le Python tout est bon'.split()
print(L)
L[2:4] = ['nouvelles', 'valeurs', 'et encore plus...', 1, 2, 3]
print(L)
```

Le slicing peut être utilisé sur des chaînes de caractères.

```
alphabet = 'abcdefghijklmnopqrstuvwxyz'
print(alphabet[:4], alphabet[-4:], alphabet[0:12:3])
```

> **Exercice**:
>
> 1. Découpez l'alphabet en deux parties égales
> 2. Prenez une lettre sur deux

```
# Votre code ici
```

```
# Solution
alphabet = "abcdefghijklmnopqrstuvwxyz"

print(alphabet[:len(alphabet) // 2])
print(alphabet[len(alphabet) // 2:])
print(alphabet[::2])
```

## Chaînes de caractères, le retour

Les chaînes de caractères sont des séquences immuables donc on les manipule comme telles.

```
# Cette cellule génère une erreur
non_mutable = 'abcdefgh'
non_mutable[3] = 'D'
```

Il faut construire une nouvelle chaîne de caractère. En concaténant des morceaux (slices) de la chaîne originale:

```
nouvelle_chaine = non_mutable[:3] + 'D' + non_mutable[4:]
print(nouvelle_chaine)
```

Ou alors en utilisant une transformation en liste, puis a nouveau en chaîne:

```
a = list(non_mutable)
a[3] = 'D'
print(''.join(a))
```

On peut savoir si une chaîne se trouve dans une autre

```
print('123' in 'azerty_123_uiop')
print('AZE' in 'azerty_123_uiop')
print('aze' in 'azerty_123_uiop')
```

La longueur d'une chaîne s'obtient avec `len()`.

```
print(len(non_mutable))
```

> **Exercice**, dans la cellule ci-dessous:
>
> 1. **Insérez** le caractère '#' au milieu de la chaîne donnée.
> 2. Idem mais **coupez** la chaine en 3 parties, et insérez le caractère '@' entre chacune d'elles.
> 3. **Insérez** le caractère '|' entre chaque caractère de la chaîne.

```
chaine_donnee = 'azertyuioppoiuytreza'
# Votre code ici
```

```
# Solution
chaine_donnee = "azertyuioppoiuytreza"

milieu = len(chaine_donnee) // 2
print(chaine_donnee[:milieu] + '#' + chaine_donnee[milieu:])

intervale = len(chaine_donnee) // 3
print(chaine_donnee[:intervale] + '@' + chaine_donnee[intervale:2 * intervale + 1] + '@' + chaine_donnee[2 * intervale

print('|'.join(chaine_donnee))
```

## Les dictionnaires

Les dictionnaires ou listes associatives sont des conteneurs où les objets ne sont **pas** ordonnés ni accessibles par un indice mais s associés à une clé d'accès.

L'accès aux éléments se fait comme pour les listes ou tuples, avec les `[]`.

```
dico = {cle1: valeur1, cle2: valeur2, ...}
```

Les clés peuvent avoir n'importe qu'elle valeur à condition qu'elles soient de type immuable : les listes ne peuvent pas servir de clés les chaînes de caractères et les tuples le peuvent.

Dans `dico`, on accède à `valeur1` avec la syntaxe `dico[cle1]`.

### Un exemple

*↳ 6 cellules masquées*

### Un autre exemple

*↳ 11 cellules masquées*

## Les ensembles

La fonction `set()` permet de créer des ensembles. Les ensembles sont des conteneurs qui n'autorisent pas de duplication d'objets contrairement aux listes et tuples.

On peut créer des ensembles de cette façon :

```
ensemble = set(iterable)
```

Où `iterable` peut être n'importe quel objet qui supporte l'itération : liste, tuple, dictionnaire, un autre set (pour en faire une copie), v objets itérables, etc...

Tout comme pour les dictionnaires, l'opérateur `in` est efficace.

### Exemples

```
list1 = [1, 1, 2]
tupl1 = ('un', 'un', 'deux', 1, 3)

b = set(list1)
a = set(tupl1)
```

```
print(a, b)

print("La chaîne 'un' est elle dans l'ensemble ?", 'un' in a)
a.remove('un')
print("La chaîne 'un' est-elle toujours dans l'ensemble ?", 'un' in a)

print(a, b)
```

Des opérations supplémentaires sont possibles sur des ensembles. Elles sont calquées sur les opérations mathématiques :

- union
- intersection
- etc...

```
print('intersection :', a & b)
print('union :', a | b)
```

Pour plus d'informations sur les ensembles, voir ici

# Fichiers

## Ouverture

L'instruction :

```
f = open('interessant.txt', mode='r')
```

ouvre le fichier `interessant.txt` en mode lecture seule et le renvoie dans l'objet `f`.

- On peut spécifier un chemin d'accès complet ou relatif au répertoire courant
- Le caractère de séparation pour les répertoires peut être différent en fonction du système d'exploitation (`/` pour unix et `\` pou
  voir le module `os.path` ou mieux la bibliothèque `pathlib`.

### Modes d'ouverture communs

- `'r'` : lecture seule
- `'w'` : écriture seule
- `'a'` : ajout à partir de la fin du fichier

**Note :** Avec `'w'` et `'a'`, le fichier est créé s'il n'existe pas.

Pour plus d'informations sur les objets fichiers, voir ici, pour la documentation de la fonction `open()`, voir là.

## Fermeture

On ferme le fichier `f` avec l'instruction :

```
f.close()
```

# Méthodes de lecture

- `f.read()` : retourne tout le contenu de `f` sous la forme d'une chaîne de caractères.

```
from pathlib import Path  # Pour la compatibilité avec Windows
f = open(Path('texte.txt'), mode='r')
texte = f.read()
print('"texte" est un objet de type', type(texte), 'de longueur', len(texte), 'caractères:')
print(texte)
print('Contenu en raw string:')
print(repr(texte))
f.close()
```

- `f.readlines()` : retourne toutes les lignes de `f` sous la forme d'une liste de chaînes de caractères.

```
f = open('exos/interessant.txt', mode='r')
lignes = f.readlines()
```

```
print('"lignes" est un objet de type', type(lignes), 'contenant', len(lignes), 'éléments:')
print(lignes)
f.close()
```

## Méthodes d'écriture

- `f.write('du texte')` : écrit la chaine `'du texte'` dans `f`

```
chaine = 'Je sais écrire\n...\navec Python !\n'
# mode 'w' : on écrase le contenu du fichier s'il existe
# encoding UTF-8: on peut écrire des chaînes avec des caractères non ASCII
f = open('texte.txt', mode='w', encoding='utf8')
f.write(chaine)
f.close()
```

**Note :** du point de vue du système, rien n'est écrit dans le fichier avant l'appel de `f.close()`

- `f.writelines(ma_sequence)` : écrit la séquence `ma_sequence` dans `f` en mettant bout à bout les éléments

```
sequence = ['Je sais ajouter\n', 'du texte\n', 'avec Python !\n']
f = open('texte.txt', mode='a')
# mode 'a' : on ajoute à la fin du fichier
# encoding par défaut: uniquement des caractères ASCII
f.writelines(sequence)
f.close()
```

```
chaine = 'Pour ajouter du texte avec des caractères accentués: ï\n'
f = open('texte.txt', mode='a', encoding='utf8')
# mode 'a' : on ajoute à la fin du fichier
# encoding UTF-8: on peut écrire des chaînes avec des caractères non ASCII
f.write(chaine)
f.close()
```

> **Exercice :**
>
> 1. écrire le contenu de la liste `mystere` dans le fichier `coded.txt` puis fermer ce dernier
> 2. lire le fichier `coded.txt` et le stocker dans une chaine `coded`
> 3. Décoder la chaîne `coded` avec les instructions suivantes:
>
> ```
> import codecs
> decoded = codecs.decode(coded, encoding='rot13')
> ```
>
> 4. écrire la chaine `decoded` dans le fichier `decoded.txt` et fermer ce dernier
> 5. visualiser le contenu du fichier `decoded.txt` dans un éditeur de texte

```
mystere = ['Gur Mra bs Clguba, ol Gvz Crgref\n\n',
           'Ornhgvshy vf orggre guna htyl.\n',
           'Rkcyvpvg vf orggre guna vzcyvpvg.\n']
# Votre code ci-dessous
```

```
# Solution

# Cette ligne permet l'utilisation de codecs.decode(), c.f. ligne 21
import codecs

mystere = ["Gur Mra bs Clguba, ol Gvz Crgref\n\n",
           "Ornhgvshy vf orggre guna htyl.\n",
           "Rkcyvpvg vf orggre guna vzcyvpvg.\n"]

# On écrit ces chaînes de caractères dans le fichier "coded.txt"
f = open('coded.txt', mode='w')
f.writelines(mystere)
f.close()

# On relit le contenu du fichier "coded.txt" que l'on vient de créer
f = open('coded.txt', mode='r')
coded = f.read()
f.close()

# On décode le message mystérieux
decoded = codecs.decode(coded, encoding='rot13')

# On écrit le message décodé dans un autre fichier
f = open('decoded.txt', mode='w')
f.write(decoded)
f.close()
```

```
# Vous pouvez ensuite aller consulter le message décodé dans le fichier "decoded.txt"
%cat decoded.txt
```