



# Formation Python – Initiation



# Sommaire

Table des matières



**Introduction au langage Python**  
Les bases du langage Python



**Les types de données de base**  
Les entiers, les décimaux, les booléens, les chaînes de caractères



**Les structures conditionnelles**  
Les conditions if elif else pour les tests d'hypothèses



**Les boucles**  
Les boucles for et while



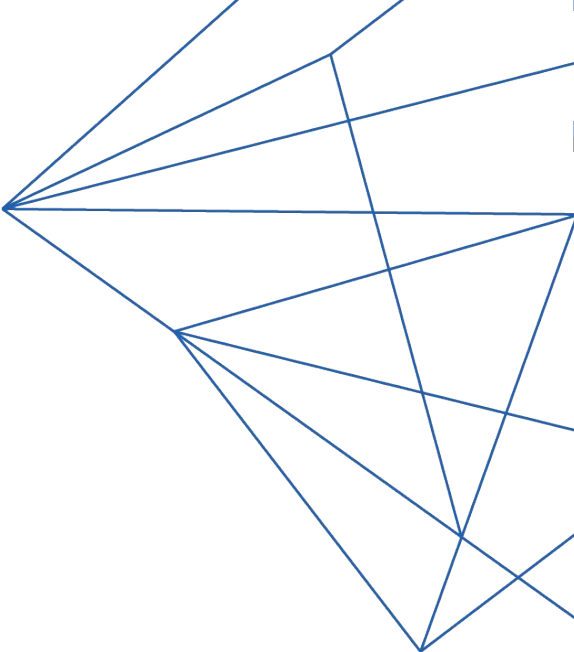
**Les fonctions**  
Définir une fonction  
Les fonctions lambda



**Le langage orienté objet**  
Introduction aux classes et méthodes



**Introduction aux modules python**



# Introduction au langage python

---



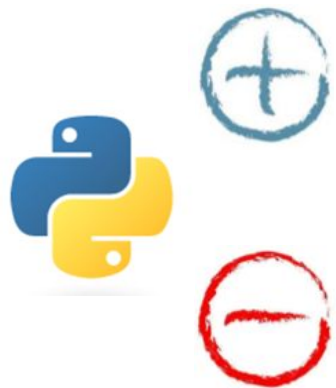


- Un langage de programmation développé en 1991
- Un langage **open source**
- Une **communauté d'utilisateurs** très importante et active
- Un langage **puissant** et **simple à apprendre**
- Un langage **modulaire** (basé sur des modules ou packages)
- De nombreuses applications possibles sur python
  - Développer des interfaces graphiques,
  - Créer des interfaces web (avec Django par exemple),
  - Créer des applications et jeux,
  - Analyser des données, réaliser des modélisations

Python est un **langage de script interprété**, contrairement à JAVA ou au langage C qui sont des langages dits compilés

Un langage compilé nécessite de lancer un programme spécifique – le compilateur – avant l'exécution d'un programme

- Cette étape, appelée compilation, consiste en la transcription du code en langage machine



Un langage plus simple: pas nécessaire de compiler le code après chaque modification de code

Une meilleure portabilité sur différents OS

Un temps d'exécution plus long

2 versions de Python sont couramment utilisées:

- La version 2.7
- La version 3

Ces deux versions comportent des syntaxes distinctes sur certaines fonctions, bien que les différences soient mineures.

Nous utiliserons dans cette formation la **version 3.7.2**. La version 2.7 ne sera bientôt plus maintenue.

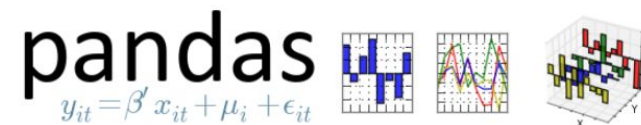
Pas de Begin / End en python pour délimiter le code (comme en Pascal/Ruby) , ni d'accolades comme en C/C++. Python nécessite une **indentation systématique** du code, que nombre d'éditeurs de code génèrent automatiquement.

Python est **sensible à la casse**. Les mots clés comportent parfois des majuscules, qui doivent être respectés, de même pour les variables déclarées

Une **distribution** est un ensemble de logiciels/modules prêts à installer.  
Il existe plusieurs distributions permettant de développer et exécuter du code Python.  
Elles embarquent un certain nombre de packages, ainsi qu'un éditeur de code

La principale distribution

- Anaconda: <https://www.continuum.io/Downloads>



2 solutions pour coder et exécuter du code python:

- Les IDE (environnement de développement)



- Les **notebooks**: environnement interactif, se présentant sous forme d'une page web. Il y a possibilité d'insérer du code, des formules mathématiques en LaTeX, des graphiques...
  - Le plus connu est iPython, appelé plus généralement **Jupyter** (<http://jupyter.org/install.html>).

Il est nativement installé à partir d'Anaconda





## Exercice

Instructions	Résultats
1	1
1+1	2
2*5	10
bonjour	<b>NameError:</b> name 'bonjour' is not defined
'Bonjour'	'Bonjour'
a = 2 a	2
print (a)	2

## Conclusions :

- ❑ Python = calculateur
- ❑ Les chaines de caractères doivent être encapsulées entre des côtes, simples, doubles ou triples. Sinon, il s'agit de variables, fonctions ou objets
- ❑ Le signe = sert à affecter une valeur à une variable
- ❑ Lorsque l'on fait appel à une variable, il faut que celle-ci ait été affectée au préalable
- ❑ La fonction **print** sert à afficher du texte ou la valeur d'une variable dans la console

Une variable, c'est un **symbole** défini par un **nom** et une **valeur**.

Les variables sont dites **dynamiques**: leur valeur peut changer au cours du temps

Une variable est aussi définie par un **type**, lui aussi **dynamique** (contrairement à C ou JAVA)

Pour **assigner une valeur** à une variable, on utilise le **signe =**

En python, le nom d'une variable:

- Peut contenir des lettres, des chiffres, ainsi que le underscore \_ (pas d'espace ni caractères spéciaux)
- Ne peut commencer par un chiffre
- N'est pas limité en longueur
- Par convention, une variable commence par une minuscule



Python est sensible à la casse, respectez bien les majuscules

Certains **mots clés** sont **réservés**: il n'est pas possible de nommer une variable ou un objet avec un nom de la liste ci-dessous

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

- L'affectation simple  $a = 1$
- L'affectation multiple  $x = y = 1$
- L'affectation simultanée  $a, b = \text{'Bonjour'}, 1$
- Il existe une notation régulièrement utilisée lorsqu'il s'agit d'incrémenter un compteur

```
a = a + 1  
a = a - 1  
a = a * 2  
a = a / 3
```



```
a += 1  
a -= 1  
a *= 2  
a /= 3
```

- Il est possible de permuter simplement des variables

```
a, b = b, a
```

Comme dans tout langage informatique, il est important de **documenter son code** et de **l'indenter** afin de le rendre lisible et facilement débuggable. Certaines distributions nous simplifient la tâche en indentant automatiquement le code, comme c'est le cas de Spyder.



Dans certains cas, l'indentation automatique ne fonctionne pas correctement, ce qui peut générer des erreurs dans la console

Pour insérer du commentaire dans le code, on utilise le symbole #.  
Il faudra insérer ce symbole pour chaque ligne de commentaire.

Il existe aussi des **docstrings** (pour documentation string), utiles pour documenter son code en y indiquant l'objectif du programme, son auteur, la date de release, les variables en entrée et les résultats.

```
"""
```

```
Ceci est un exemple de  
docstring
```

```
Auteur : F. STEVENOOT  
"""
```

- ❑ Les docstrings sont identifiés par des triples guillemets
- ❑ Les lignes entre le début et la fin des docstrings ne nécessitent pas de guillemet

# Les types de données de base

---



Il existe deux types de données numériques:

- Les entiers ou **int**,
- Les nombres décimaux ou **float**



Pour déterminer le type d'un objet, d'une variable, ou du résultat d'un calcul, on utilise la fonction `type( )`

Instructions	Résultats
<code>type(1)</code>	int
<code>type(-2)</code>	int
<code>type(5.0)</code>	float
<code>type(5/2)</code>	float
<code>type(5/2.0)</code>	float



En version 2, lorsque l'on **divise deux entiers**, le résultat est un int.

Les opérateurs classiques s'appliquent de la même manière que dans la plupart des langages.

Opérateurs Arithmétiques	
+	Addition
-	Soustraction
*	Multiplication
/	Division
//	Partie entière du quotient de la division
%	Modulo (reste de la division)
**	Puissance

Il existe deux fonctions pour convertir une variable en entier, ou en flottant:

- La fonction **int**( ) convertit en entier. Equivaut à la partie entière d'un nombre décimal
- La fonction **float**( ) convertit une variable en flottant
- Les fonction **int**() et **float**() permettent d'initialiser des variables a 0 pour un entier et 0.0 pour un nombre décimal



Un booléen est une variable à deux états, résultat de conditions logiques (ou tests).  
Les valeurs prises par une variable booléenne sont **True** et **False** en python.



Ne pas oublier que python est sensible à la casse. Attention aux majuscules.

Le **type** renvoyé par python est **bool**. La fonction **bool()** permet de convertir une variable en booléen.

## Opérateurs de comparaison

==

Test d'égalité

!=

Différence

<= , < , >= , >

Comparaison

## Opérateurs logiques

and

Si les deux membres ont vrai alors vrai sinon faux

or

Si un des deux membres est vrai alors vrai sinon faux

not

L'inverse de la condition

## Exemple:

Instructions	Résultat
True == False	False
True > True	False
True < False	False
True > False	True
b = (2 != 3) b	True
a = 5 b = 3	False
(a < b) and (b < a)	
(a < b) or (b < a)	True
not(b != a)	False



La condition `int(True) == 1` est vraie, de même que `int(False) == 0`

Les **chaines de caractères** (string en anglais) sont constituées d'une séquence de lettres/nombres/caractères spéciaux

Le type des variables chaines de caractères est **str** en python

Pour convertir une variable en chaine de caractères, ou initialiser une chaine vide, on utilise la fonction *str()*

Une chaine de caractères est identifiée par:

- Des simples cotes ' '
- Des doubles cotes " "
- Des triples cotes '''' '''



Lorsque votre chaine de caractères comprend des apostrophes, cela peut poser problème lorsque l'on utilise les simples cotes. On utilise alors l'antislash \ pour échapper l'apostrophe

```
chaine = 'J'aime Python'  
=> SyntaxError: invalid syntax
```



```
chaine = 'J\'aime Python'  
OU  
chaine = "J'aime Python"
```

Le **slicing** consiste à **découper** une séquence en **sous éléments**.

Dans le cas de chaines de caractères, le slicing revient à découper les chaines en caractères unitaires

Pour ce faire, on utilise des **indexes** pour repérer la position du caractère dans la chaine, que l'on passe entre crochets []

## Exemple:

Soit A une chaine contenant l'ensemble des lettres de l'alphabet  
Si l'on souhaite la première lettre de l'alphabet:

```
A = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'  
print (A[0])
```



En python, les indexes commencent à 0

Pour récupérer **plusieurs caractères** compris **entre deux bornes**, on utilise les **:** pour séparer les deux indexes

## Exemple:

Soit A une chaine contenant l'ensemble des lettres de l'alphabet

Si l'on souhaite récupérer une chaine avec les 3 premières lettres de l'alphabet:

```
A = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'  
print (A[0 : 3])
```



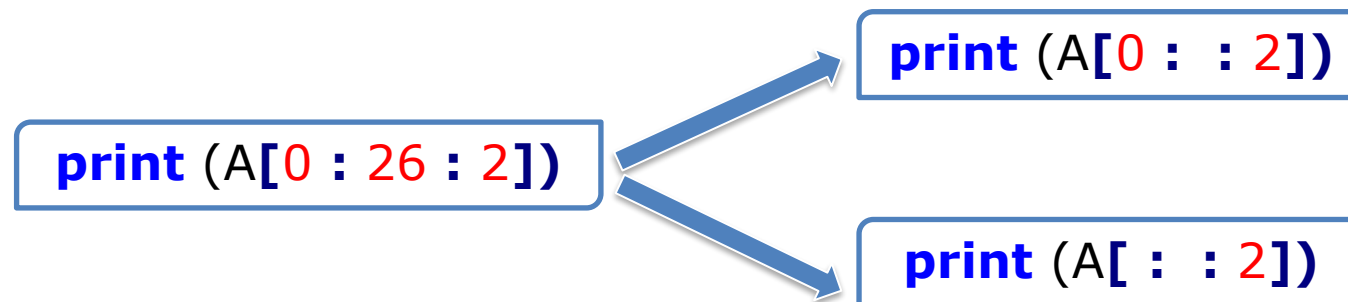
La borne supérieure est exclue, la borne inférieure est incluse

Il est possible de définir un pas pour le slicing pour récupérer par exemple les lettres d'indice pair

```
print (A[0 : 26 : 2])  
=> 'ACEGIKMOQSUY'
```

Le pas est la 3<sup>e</sup> composante, séparée par **:**  
de la borne sup

Il n'est pas nécessaire de spécifier l'ensemble des indices.



Il est possible de « renverser » une chaîne de caractère, en utilisant un pas de -1

```
A = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
print (A[ : : -1])
=> 'ZYXWVUTSRQPONMLKJIHGFEDCBA'
```

Pour déterminer la **longueur d'une chaîne** de caractères, on utilise la fonction **len()**

```
len(A)
=> 26
```

Il existe quelques opérateurs sur les chaines de caractères pour réaliser des opérations simples

- La **concaténation** : on utilise l'opérateur +

'Bonjour' + 'à tous'  
=> 'Bonjour à tous'

- La **répétition**: on utilise l'opérateur \*

'Bonjour' \* 3  
=> 'BonjourBonjourBonjour'

- L'**appartenance**: on utilise l'opérateur in

'jour' in 'Bonjour'  
=> True

Il est souvent utile de pouvoir paramétrer une chaîne de caractère lorsque l'on print un résultat. Plusieurs possibilités s'offrent à vous :

- La plus simple : la concaténation de chaînes de caractères à l'aide du +

```
age = 36
prenom = "Fabien"

print("Je m'appelle " + prenom + " et j'ai " + str(age) + "ans")
```

- L'utilisation de la virgule pour afficher la valeur de plusieurs objets

```
print("Je m'appelle" , prenom , "et j'ai" , age , "ans")
```

- La méthode format

```
print("Je m'appelle {} et j'ai {} ans".format(prenom, age))
```

On utilise les {} pour spécifier qu'un paramètre est attendu.  
Les paramètres sont affichés dans l'ordre d'apparition de la méthode format  
Il est possible de définir un ordre différent en spécifiant un index dans les {}

```
print("Je m'appelle {1} et j'ai {0} ans".format(age, prenom))
```



list = **séquence** d'éléments de **type identique ou différents**.

Son type est **list**

Une liste est identifiée par des **crochets** []

Pour affecter une liste vide dans une variable, on utilise la fonction **list()** ou []

Le **slicing** fonctionne de manière identique sur les listes que sur les chaînes de caractères

## Exemple:

```
liste_vide1 = []  
liste_vide2 = list()  
liste_nombres = [1,2,3,4,5]  
liste_animaux = ['chat', 'chien', 'poisson']  
liste_diff = [1, True, 'tomate', [1, 2, 3]]
```

```
print (liste_animaux[1])
```

=> 'chien'



De manière identique aux strings, on peut connaître le nombre d'éléments d'une liste en utilisant la fonction **len(liste)**

Il existe des opérateurs sur les listes pour réaliser des opérations simples

- La concaténation : on utilise l'opérateur +

```
[1,2] + ['a']  
=> [1,2,'a']
```

- La **répétition**: on utilise l'opérateur \*

```
[1,2] * 3  
=> [1,2,1,2,1,2]
```

- L'**appartenance**: on utilise l'opérateur in

```
1 in [1,2]  
=> True
```

Pour supprimer un élément d'une liste, on utilisera l'instruction **del** en spécifiant l'index de l'élément à supprimer

```
liste = [1, 2, 3, 'carotte', [1, 2]]  
del liste[3]  
print(liste)
```

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

Pour effectuer une opération simple sur une liste, on peut utiliser les **compréhensions de liste**.

Avantages :

- Tient en 1 ligne
- Lisibilité accrue

**Exemple :**

```
liste = [1, 2, 3, 4, 5, 6]  
[element for element in liste]
```

⇒ [1, 2, 3, 4, 5, 6]

```
[element-1 for element in liste if element > 3]
```

⇒ [3, 4, 5]

Les **tuples** sont un autre type de données fréquemment utilisé.

- Sur un tuple, la fonction **type()** renvoie le type **tuple**
- Un tuple est identifié par des **parenthèses** ()
- Comme les listes, il s'agit d'une **séquence d'éléments de type identique ou différents**
- L'un des exemples les plus courants correspond aux coordonnées de points: latitude/longitude, coordonnées spatiales x/y/z
- Le **slicing** fonctionne de manière identique aux strings et listes

```
tuple_vide = tuple()  
tuple_vide2 = ()  
coordonnees = (2.5, 3.5)  
tuple_mixte = ('tomate', 1)  
print ('coordonnée x = ', coordonnees[0])  
=> coordonnée x = 2.5
```



De manière identique aux strings, on peut connaître le nombre d'éléments d'un tuple en utilisant la fonction **len(tuple)**

Il existe des opérateurs sur les tuples pour réaliser des opérations simples

- La **concaténation** : on utilise l'opérateur **+**

$$(1,2) + (1,2,3,a) \\ \Rightarrow (1,2,1,2,3,a)$$

- La **répétition**: on utilise l'opérateur **\***

$$(1,2) * 3 \\ \Rightarrow (1,2,1,2,1,2)$$

- L'**appartenance**: on utilise l'opérateur **in**

$$1 \text{ in } (1,2) \\ \Rightarrow \text{True}$$

Les **dictionnaires** sont composés de couples permettant d'associer des **clés** à des **valeurs** et sont de type **dict**

- Les dictionnaires sont identifiés par des accolades {}
- Pour initialiser un dictionnaire :

```
dictionnaire_vide = dict()  
dictionnaire_vide2 = {}
```

```
dictionnaire = {'pommes': 3, 'poires': 5, 'carottes': 6}
```

**ou**

```
dictionnaire = {}  
dictionnaire['pommes'] = 3  
dictionnaire['poires'] = 5  
dictionnaire['carottes'] = 6
```

Le schéma est *clé : valeur associée*

Le slicing ne fonctionne pas sur les dictionnaires. Pour récupérer la valeur correspondant à une clé:

```
print(dictionnaire['pommes'])  
=> 3
```

Les **clés** et **valeurs** peuvent être **de tout type** (int, float, bool, string)  
Les clés et les valeurs peuvent être de **type différents**

La fonction **len()** renvoie le **nombre de clés distinctes**

Si un dictionnaire est créé avec **>2 clés égales**, alors Python ne gardera que la **dernière clé**



# Les structures conditionnelles

---



Les structures conditionnelles permettent de **tester ou une plusieurs conditions**, et d'exécuter une ou plusieurs instructions différentes selon les cas.

La forme la plus simple consiste à ne tester qu'une seule condition. La structure est la suivante:

**if** *condition 1 est vraie:*  
*instruction 1*

Pas de then en Python: l'instruction if se termine par des :

L'instruction exécutée est identifiée par une indentation automatique

```
a = 2
if a > 0 :
    print('a est positif')
```

Il est possible de tester des combinaisons de conditions à l'aide de **and** et **or**

```
a = 2
b = 3
if a > 0 and b > 0:
    print('a et b sont positifs')
```

On qualifie l'ensemble des instructions de **bloc** (qu'il y en ait une ou plusieurs)

Il est aussi possible d'exécuter plusieurs instructions si la condition testée est vraie:

```
a = 2
if a > 0 :
    print('a est positif')
    a -= 1
```



La valeur de a sera alors égale à 1 en sortie de la boucle

Lorsque l'on souhaite tester plusieurs conditions, on peut utiliser la forme complète des structures conditionnelles qui se présente sous la forme :

```
if condition 1 est vraie:  
    instruction 1  
elif condition 2 est vraie :  
    instruction 2  
else:  
    instruction 3
```

## Exemple:

```
if a > 0:  
    print('a est positif')  
elif a == 0:  
    print('a est nul')  
else:  
    print('a est négatif')
```

Soit a et b deux nombres en entrée.

Ecrire une structure conditionnelle de telle sorte que l'on renvoie :

- 'a et b sont pairs' dans le cas où les deux nombres sont pairs
- 'a et b sont impairs' dans le cas où les deux nombres sont impairs
- 'a est pair, b est impair'
- 'a est impair, b est pair'

```
if a%2 == 0 and b%2 == 0:  
    print('a et b sont pairs')  
  
elif a%2 == 1 and b%2 == 1:  
    print('a et b sont impairs')  
  
elif a%2 == 0 and b%2 == 1:  
    print('a est pair, b est impair')  
  
else:  
    print('a est impair, b est pair')
```

# Les boucles

---



La première boucle que nous allons utiliser est la boucle **while**.

La syntaxe est la suivante:

```
while condition est vraie:  
    instruction 1  
    instruction 2
```

De même que pour les structures conditionnelles if, les boucles while se termine par **:** suivi d'une **indentation**

## Exemple:

```
i = 10  
print('Ceci est un compte à rebours')  
while i >= 0:  
    print(i)  
    i -= 1  
print('Bonne année!')
```

Il faut **incrémenter le compteur** à chaque itération, ici avec un pas de -1

Il faut **déclarer** au préalable **le compteur** et sa valeur



Repartons de notre exercice sur les structures conditionnelles.  
Pour chaque nombre de 1 à 10, tester si il est pair ou impair et afficher le résultat sous la forme:

*le nombre 1 est un impair*

```
i = 1
while i <= 10:
    if i % 2 == 0:
        print('le nombre', i, 'est pair')
    else:
        print('le nombre', i, 'est impair')
    i += 1
```

La boucle **for** est une boucle particulière en python, en ce sens où elle travaille **sur des séquences**.

Nous avons vu précédemment plusieurs types de séquences: les strings, les listes et les tuples.  
La syntaxe de la boucle for est la suivante:

```
for élément dans une séquence:  
    instruction 1  
    instruction 2
```

## Exemple:

Pour récupérer un élément dans une séquence, on utilise l'opérateur **in**

```
a = 'bonjour'  
for lettre in a:  
    print(lettre)
```



b  
o  
n  
j  
o  
u  
r



Il n'est pas nécessaire d'incrémenter l'élément dans la boucle

Généralement, on utilise des indexes numériques lorsque l'on utilise les boucles. Dans la mesure où la boucle **for** travaille sur des séquences, la première solution serait d'utiliser une liste numérique:

```
liste_index = [0, 1, 2, 3, 4, 5, 6]
a = 'bonjour'
for index in liste_index:
    print (a[index])
```



Il n'est pas très pratique de devoir initialiser une liste d'indexes avant d'opérer une boucle. Dans ce cas, on utilise la fonction **range()**

La fonction **range()** génère une **liste séquentielle d'éléments numériques**.

La syntaxe est la suivante:

```
range(start, stop, pas)
```

- **start** : l'indice de début. Ce paramètre est **optionnel** et vaut 0 par défaut. Elle est **incluse dans la liste**
- **stop** : l'indice de fin. Ce paramètre est **obligatoire**. Cette borne **exclue de la liste**
- **pas** : le pas d'incrément. Ce paramètre est **facultatif**

Voici quelques exemples d'utilisation de la fonction **range()**

```
range(5)
    ⇒ [0, 1, 2, 3, 4]
range(1, 3)
    ⇒ [1, 2]
range(5, 2)
    ⇒ []
range(0, 5, 2)
    ⇒ [0, 2, 4]

a = 'bonjour'
range(len(a))
    ⇒ [0, 1, 2, 3, 4, 5, 6]
```



En **Python 3**, la fonction **range** renvoie un *objet de type range* (équivalent à une liste). En python 2, c'est une liste qui est renvoyée

Il n'est pas possible d'utiliser un pas sans déclarer la borne inf ET la borne sup

**Exemple:**

```
a = 'bonjour'
for index in range(len(a)):
    print(a[index])
```

Repartons de notre exercice sur les structures conditionnelles.  
Pour chaque nombre de 1 à 10, tester si il est pair ou impair et afficher le résultat sous la forme:

*le nombre 1 est un impair*

```
for i in range(1, 11):  
    if i % 2 == 0:  
        print('le nombre', i, 'est pair')  
    else:  
        print('le nombre', i, 'est impair')
```

Il est possible de contrôler l'exécution d'une boucle, par exemple en forçant son interruption lorsqu'une condition est remplie.

On utilise alors l'instruction **break**

Dans notre exercice précédent:

```
for i in range(1,11):  
    if i % 2 == 0:  
        print('le nombre', i, 'est pair')  
    else:  
        print('le nombre', i, 'est impair')  
    if i == 5:  
        break
```

La boucle s'arrête alors lorsque le nombre est 5



L'instruction break peut être utilisée aussi bien avec la boucle **for** qu'avec la boucle **while**



L'instruction **continue** permet de retourner au début de la boucle sans exécuter les instructions suivantes.

En revanche, la boucle n'est pas stoppée.

Dans notre exercice précédent:

```
for i in range(1,11):  
    if i == 5:  
        continue  
    if i % 2 == 0:  
        print('le nombre', i, 'est pair')  
    else:  
        print('le nombre', i, 'est impair')
```

Seule l'itération lorsque i vaut 5 n'est pas exécutée



L'instruction continue peut être utilisée aussi bien avec la boucle **for** qu'avec la boucle **while**

Il existe une dernière instruction utile lorsque l'on travaille sur les boucles: l'instruction **pass**

Cette instruction est utilisée lorsque une instruction est requise syntaxiquement, mais que l'on souhaite pas exécuter d'instruction spécifique.

Par exemple, lorsque l'on sait qu'une condition est à tester, mais que l'on veut y revenir plus tard

# Les fonctions

---



Une fonction est un bloc d'instructions permettant de réaliser une action, par exemple pour renvoyer une valeur.

- Le principal avantage d'une fonction est qu'elle **peut être réutilisée très simplement** une fois définie.
- Une fonction commence par le mot-clé **def** suivi du **nom de la fonction** et de **parenthèses**.
- Il est possible de passer des **paramètres optionnels** à l'intérieur des paramètres.
- L'instruction de déclaration de la fonction se termine par :
- Une fonction se termine par le mot clé **return**, qui permet de renvoyer une **valeur optionnelle** pour clore la fonction (optionnel mais conseillé)
- Les instructions à l'intérieur de la fonction sont **indentées**
- Bien que facultatives, il est conseillé de documenter les fonctions à l'aide de **docstrings**

La syntaxe est la suivante

```
def ma_fonction(parametre1 , paramètre2...):  
    """  
    cette fonction renvoie...  
    """  
    instruction1  
    instruction2  
    return xxx
```

*Exemple :*

```
def affichage(chaine = 'Bonjour'):  
    """  
    cette fonction renvoie la chaine  
    passée en paramètre  
    """  
    return(chaine)
```



Le « return » peut être fait avec ou sans parenthèses

La définition d'une fonction ne sert qu'à lui donner un nom. à définir les blocs d'instructions ainsi que les paramètres.

Pour faire appel à une fonction, il suffit d'écrire son nom dans la console (ou dans une autre fonction dans le cas de fonctions imbriquées ) avec la valeur des paramètres souhaitée.

```
affichage()
```

⇒ Bonjour

```
affichage('Bienvenue à la formation python')
```

⇒ Bienvenue à la formation python



Le fait d'avoir affecté une valeur au paramètre lors de la déclaration de la fonction implique que ce paramètre sera utilisé par défaut

Définir une fonction **aire()** qui renvoie la surface d'un rectangle de longueur L et de largeur l

```
def aire(L , l):
```

```
    """
```

```
    cette fonction renvoie l'aire du rectangle de  
    longueur L et de largeur l
```

```
    """
```

```
return L * l
```



Il existe différents types de paramètres:

- les paramètres obligatoires

```
def affichage(chaine):  
    print(chaine)
```

```
affichage()
```

=> TypeError: affichage() takes exactly 1 argument (0 given)

Erreur indiquant que le paramètre est manquant

- les paramètres « optionnels »

```
def affichage(chaine = 'bonjour'):  
    print(chaine)
```

```
affichage()
```

=> 'Bonjour'



Par défaut, les paramètres sont définis comme positionnels: il faut alors les passer dans le même ordre, à moins de préciser leur nom

Il est possible de définir des fonctions d'une manière différente et condensée: ce sont les fonctions lambda

```
def aire(L, l):  
    """  
    cette fonction renvoie l'aire du rectangle de  
    longueur L et de largeur l  
    """  
    return L * l
```

Cette fonction peut être définie de la manière suivante:

```
aire = lambda L, l : L * l  
aire(3,5)
```

On utilise le terme **lambda** pour définir la fonction. Les paramètres sont L et l  
Après les :, on écrit ce qui correspond à l'instruction **return** dans une fonction classique

Les fonctions lambda ne sont qu'une question de style, et sont utiles pour condenser le code.  
Les fonctions lambda:

- prennent un **nombre quelconque de paramètres**
- ne contiennent **qu'une seule expression**

La suite de Fibonacci est définie de la manière suivante:

$$U_0 = 0$$

$$U_1 = 1$$

$$U_2 = 1$$

$$\text{Pour tout } n > 2, U_{n+2} = U_{n+1} + U_n$$

Ecrire une fonction Fibo correspondante en python

```
def Fibo(n):  
    if n == 1:  
        return 1  
    elif n == 0:  
        return 0  
    else:  
        return Fibo(n-1) + Fibo(n-2)
```



La récursivité consiste en l'appel de la fonction elle-même dans sa définition. Le calcul des factorielles peut être défini sous la même forme.

Ecrire une fonction `facto` permettant de calculer les factorielles d'un nombre `x` de deux manières:

- avec des boucles
- avec la récursivité

**Indication :** La factorielle de 3 vaut 6, celle de 5 vaut 120

## Récurusif

```
def facto(x):  
    if x in [0,1]:  
        return 1  
    else:  
        return x * facto(x-1)
```

## avec les boucles

```
def facto2(x) :  
    resultat = 1  
    while x > 1 :  
        resultat = resultat * x  
        x = x-1  
    return resultat
```

Lorsque des fonctions ou variables sont définies dans l'interpréteur, elles n'existent que lors de la session courante et sont purgées lorsque vous redémarrez Python. Pour améliorer la réutilisabilité des fonctions que vous avez définies, il convient alors de les **enregistrer dans des programmes**, appelés **scripts**.

Il est conseillé que chaque script commence avec le code :

```
# coding: utf-8
```



On peut utiliser tout type d'encodage. Utf-8 est cependant conseillé

Python offre donc la possibilité de stocker les définitions des fonctions dans un script particulier, permettant de faire appel simplement à des fonctions: on appelle un tel script **module**

L'extension des scripts python est **.py**

Il existe un certain nombre de modules pré-installés dans la distribution anaconda, tels que:

- math pour l'utilisation des fonctions mathématiques
- matplotlib pour créer des graphiques
- pandas pour la manipulation de dataframes (tables)
- sklearn (Scikit-learn) pour les fonctions de modélisation / machine learning



Pour importer un module, on utilise le mot-clé **import**

```
import pandas
```

Une fois le module importé:

- la fonction `dir(module)` renvoie une liste exhaustive de méthodes / fonctions
- La fonction `help(module)` permet de renvoyer les docstrings des différentes méthodes / fonctions.

Si l'on souhaite par exemple utiliser la méthode `read_csv()` du module `pandas`, il conviendra alors d'appeler la méthode de la manière suivante

```
df = pandas.read_csv(...)
```



Le fait de devoir indiquer à python le nom du module dans lequel trouver la fonction / méthode étant assez lourde, il est d'usage d'**utiliser un alias** pour nommer les modules

```
import pandas as pd  
df = pd.read_csv(...)
```

Pour définir un alias, on utilise le mot-clé **as**

Il est aussi possible, non pas d'importer l'ensemble d'un module, mais **uniquement des noms de méthodes**.

Pour ce faire, on utilise la combinaison des mots-clés **from** et **import**

```
from pandas import read_csv
```

Dans ce cas, il n'est pas nécessaire d'utiliser ni le nom du module ni un alias

```
df = read_csv(...)
```

Il est possible d'importer l'ensemble des méthodes d'un module, hormis celles commençant par un underscore

```
from pandas import *
```



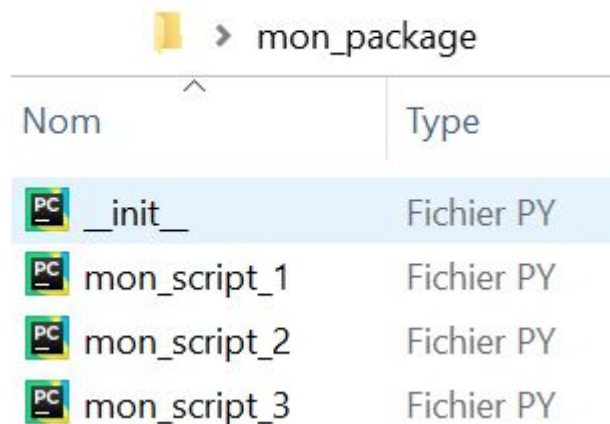
Par convention, les imports de modules sont toujours positionnés en entête des scripts

Lorsque l'on veut regrouper plusieurs module, nous pouvons créer un **package**.





Python permet de réaliser facilement un package.

Il suffit principalement de créer un répertoire contenant un script `'__init__.py'`. Ce script peut être vide.

Dans ce répertoire nous pouvons donc insérer tous les scripts/module nécessaires.



The screenshot shows a file explorer window with the title bar 'mon\_package'. Below the title bar, there is a table with two columns: 'Nom' and 'Type'. The table contains four rows of files, each with a 'PC' icon, a name, and a type 'Fichier PY'.

Nom	Type
 <code>__init__</code>	Fichier PY
 <code>mon_script_1</code>	Fichier PY
 <code>mon_script_2</code>	Fichier PY
 <code>mon_script_3</code>	Fichier PY

Par défaut, lors de l'import d'un module, python cherche dans l'ordre suivant:

- Le répertoire courant
- Si le module n'est pas trouvé, python cherche dans le répertoire défini par la variable d'environnement PYTHONPATH (définie lors de l'installation)
- Si le module n'est pas trouvé, alors la recherche s'effectue dans le répertoire par défaut: /usr/local/lib/python/ sous unix

## Exemple:

On enregistre sous test.py le code suivant

```
def bonjour():  
    print ('Bonjour!')  
    return
```

On appelle alors la fonction de la manière suivante

```
import test  
test.bonjour()
```

# Le langage orienté objet

---



**Historiquement**, les développements informatiques se basaient sur **des langages procéduraux et fonctionnels** (Fortran ou C)

Ces langages sont basés sur:

- Des procédures, qui effectuent un traitement sur les données
- Des fonctions, qui renvoie une valeur

La question posée est : que va faire mon programme?

Le **langage orienté objet** repose lui sur la notion d'objets qui contiennent:

- Des données « internes » à l'objet: on les appelle **attributs**
- Des traitements manipulant les données internes ou externes – ce sont des **méthodes**

La question posée est : de quoi est composé mon programme?

## Exemple:

**Classe  
Voiture**



**Classe  
Moto**



**Classe  
Avion**



**Classe  
Cycle**



On définit par une même **abstraction** des objets similaires ayant:

- Une même structure de données et de traitements
- Des valeurs différentes

On les appelle **Classe**.

Une classe est composée de plusieurs **membres**:

- Des **attributs**: ce sont des variables...

## Exemple:

### Classe Voiture



Les attributs peuvent être:

- Le nombre de roues
- La puissance
- La couleur
- La marque
- Le poids
- La consommation moyenne
- La contenance du réservoir

### Classe Cycle



Les attributs peuvent être:

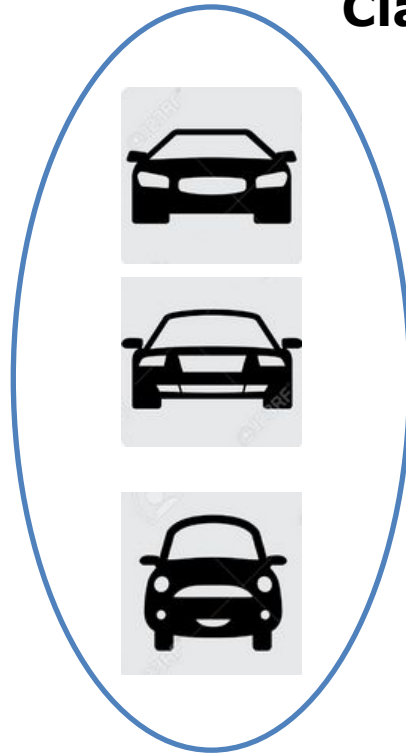
- Le nombre de roues
- La couleur
- La marque
- Le poids



... Et des **méthodes**

**Exemple:**

## Classe Voiture



Les méthodes peuvent être:

- Afficher le nombre de roues
- Calculer le nombre moyen de kilomètres avec un plein

Pour créer une classe en python, on utilise le mot clé **class**

```
class voiture():
```

```
    """
```

```
    Définition de l'objet voiture
```

```
    """
```

```
    nb_roues = 4
```

```
    def __init__(self, marque = "A déterminer"):  
        self.marque = marque
```

La fonction `__init__` est appelé **constructeur** (toujours présent): c'est une méthode

On définit ici une variable « attribut »

Le mot clé `self` (convention) fait référence à l'instance en cours

On peut ensuite **instancier** (créer) un objet, en tapant

```
Ferrari = voiture(marque = 'Ferrari')
```

On peut ensuite ajouter dans la classe des méthodes

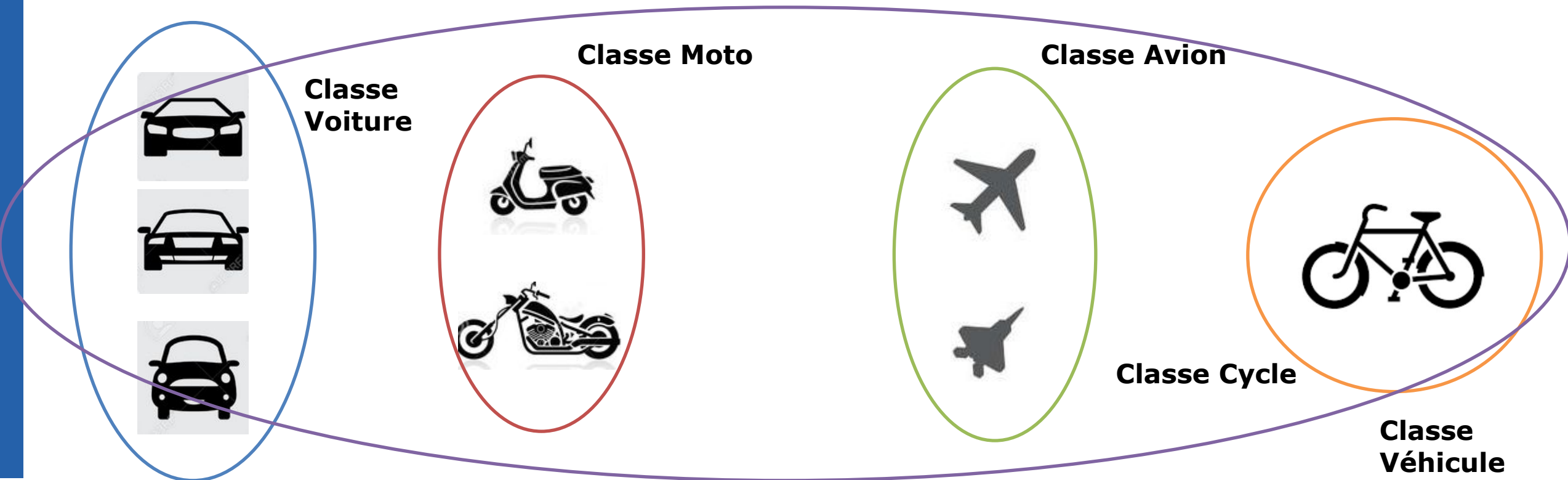
```
class voiture():  
    """  
    Définition de l'objet voiture  
    """  
    nb_roues = 4  
  
    def __init__(self, marque = "A déterminer"):  
        self.marque = marque  
  
    def get_nb_roues(self):  
        print ("Récupération du nombre de roues")  
        return self.nb_roues
```

On appellera alors la méthode sur l'objet Ferrari

```
Ferrari.get_nb_roues()  
=> 4
```

## Exemple:

L'ensemble des classes définies précédemment peuvent englobée dans une classe véhicule, car elles partagent globalement une même structure (attributs / méthodes). Les autres classes **héritent** de la classe véhicule.



```
class vehicule():
```

```
    """
```

```
    Définition de l'objet vehicule
```

```
    """
```

```
    def __init__(self, marque = "A déterminer"):  
        self.marque = marque  
        self.nb_roues = 0
```

```
    def get_nb_roues(self):  
        print ("Récupération du nombre de roues")  
        return self.nb_roues
```

```
class voiture(vehicule):
```

```
    """
```

```
    Définition de l'objet voiture
```

```
    """
```

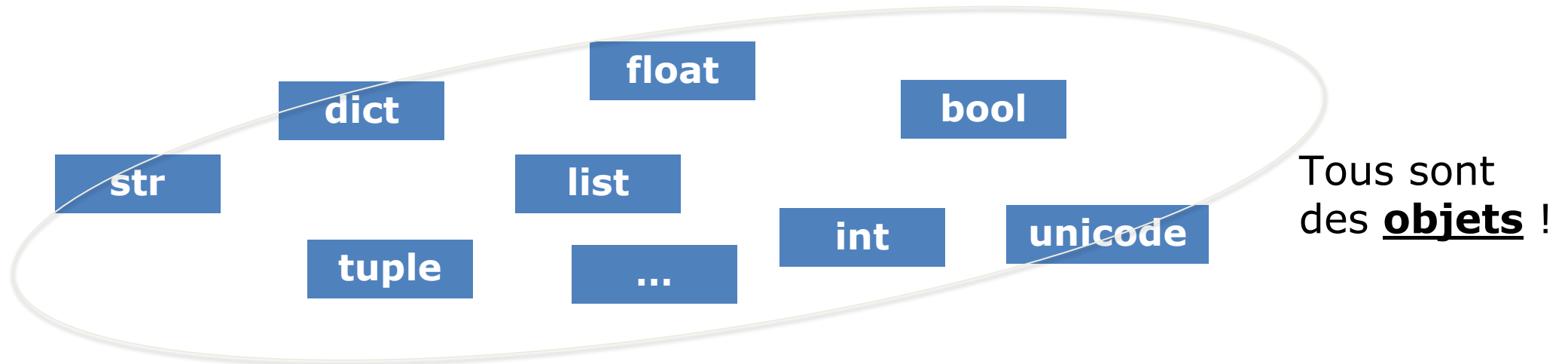
```
    nb_roues = 4
```

```
    def __init__(self, marque = "A déterminer"):  
        self.marque = marque
```

La classe voiture hérite de la méthode `get_nb_roues` de la classe `vehicule`

On définit ici l'objet dont hérite la classe

Nous avons vu brièvement comment créer (instancier) un objet en python.  
Il existe plusieurs objets prédéfinis, avec un ensemble de méthodes.



Sur ces objets, il existe une propriété importante: « **la mutabilité** »

Un objet est dit **mutable** lorsqu'il est **modifiable** après sa création, et **immutable** lorsqu'il n'est plus modifiable.

Les objets **mutables**: les listes, les dictionnaires, les objets class définis par l'utilisateur, ...

Les objets **immutables**: les int, float, bool, string, tuple, ...

Les chaînes de caractères, ou **string** sont des objets en python.

Il existe donc une **classe** string, contenant des méthodes.

L'objet string est **immutable**. Il n'est pas possible de supprimer un caractère d'une chaîne définie (pas d'instruction del), ni de modifier un élément de la chaîne.

## Exemple de méthode:

**upper()** pour mettre en majuscule

```
chaîne = 'abcd'  
chaîne.upper()  
=> 'ABCD'
```

On utilise un point pour appliquer la méthode à l'objet

On utilise les parenthèses de la même manière que pour une fonction (une méthode est une fonction avec des paramètres possibles)

## Méthodes sur les strings

<code>chaine.upper()</code> / <code>chaine.lower()</code>	Mise en majuscules / minuscules
<code>chaine.isupper()</code> / <code>chaine.islower()</code>	Renvoie True si tous les caractères sont en majuscules / minuscules
<code>chaine.isdigit()</code> / <code>chaine.isalpha()</code>	Renvoie True si tous les caractères sont numériques / des lettres
<code>chaine.count(str, beg= 0 , end=len(string))</code>	Compte le nombre d'occurrences de la sous chaine str entre beg et end
<code>chaine.find(str, beg=0 , end=len(string))</code>	Renvoie l'index de la sous chaine str entre beg et end
<code>chaine.replace(str1 , str2)</code>	Remplace la sous chaine str1 par la sous chaine str2
<code>chaine.split(str="", num=string.count(str))</code>	Découpe la chaine selon le delimiter str (différent de l'espace) et retourne une liste de sous-chaines Si num est défini, c'est le nombre maximum de sous chaines retournées
<code>chaine.join(seq)</code>	Fusionne les éléments de la liste seq à la chaine
<code>chaine.strip(str=" ")</code>	Supprime le premier caractère correspondant à str



Cette liste de méthodes n'est pas exhaustive. Vous pouvez vous référer à [l'aide de python en ligne](#)



## Exemples :

Réaliser les actions suivantes

- compter le nombre d'occurrences de la lettre l dans 'le soleil brille'

```
'le soleil brille'.count('l')
```

- remplacer les espaces par des underscores dans la chaîne 'montant en euros'

```
'montant en euros'.replace(' ', '_')
```

- créer une liste des mots de la phrase 'nous aimons apprendre python'

```
'nous aimons apprendre python'.split(' ')
```

- créer une liste des mots en majuscule à partir de la phrase 'nous aimons apprendre python'

```
[i.upper() for i in 'nous aimons apprendre python'.split(' ')]
```

Ecrire une fonction qui teste si un mot est un palindrome, et renvoie True le cas échéant  
Tester sur les chaînes:

- 'elle'
- 'kayak'
- 'eIbohPhoBiE'
- 'MoN NOm'

## **Tips :**

- Un palindrome est un mot réversible
- Ne pas tenir compte des majuscules
- Il est possible d'écrire cette fonction en une ligne

```
palindrome = lambda mot : mot.lower() == mot[::-1].lower()
```

On utilise la méthode **lower** dans chaque membre pour s'affranchir des majuscules

Pour renverser une chaîne, on peut utiliser le slicing avec un pas de -1

On affecte un nom à la lambda fonction

Les listes sont des objets python (class list).

L'objet list est **mutable**. Il est donc possible de modifier les éléments d'une liste après sa création.

Il est possible de **supprimer un élément** d'une liste en utilisant l'index correspondant avec l'instruction **del**

```
liste = [1,2,3,'carotte',[1,2]]  
del liste[3]  
print (liste)  
=> [1, 2, 3, [1, 2]]
```

Il est possible de **modifier un élément** d'une liste en utilisant l'index correspondant et l'affectation =

```
liste[3] = 4  
print (liste)  
=> [1, 2, 3, 4]
```

De la même manière, les listes sont des objets python.

Méthodes sur les listes	
<code>list.append(obj)</code>	Ajouter un objet à une liste
<code>list.count(obj)</code>	Compter le nombre d'objets dans une liste
<code>list.extend(seq)</code>	Ajouter le contenu d'une séquence à une liste
<code>list.index(obj)</code>	Renvoie le plus petit index de l'objet de la liste
<code>list.insert(off,obj)</code>	Insérer un objet dans la liste à la position off
<code>list.pop(obj)</code>	Supprime et renvoie un élément de la liste
<code>list.remove(obj)</code>	Supprime l'objet de la liste
<code>list.reverse()</code>	Renverse la liste
<code>list.sort([func])</code>	Trie la liste – func est une fonction optionnelle pour indiquer la manière de trier



Cette liste de méthodes n'est pas exhaustive. Vous pouvez vous référer à [l'aide de python en ligne](#)

## Exercice:

Réaliser les actions suivantes

- ajouter l'élément 'bananes' à la liste fruits = ['oranges', 'pommes', 'poires', 4]

```
fruits.append('bananes')
```



*['oranges', 'pommes', 'poires', 4, 'bananes']*

- insérer l'élément 'pasteques' à la liste fruits créée ci-dessus entre 'poires' et 'pommes'

```
fruits.insert(2, 'pasteques')
```



*['oranges', 'pommes', 'pasteques', 'poires', 4, 'bananes']*

- supprimer le chiffre 4 de la liste

```
fruits.remove(4)  
OU fruits.remove(fruits[4])
```



*['oranges', 'pommes', 'pasteques', 'poires', 'bananes']*

- trier la liste fruits par ordre descendant

```
fruits.sort(reverse = True)
```



*['pommes', 'poires', 'pasteques', 'oranges', 'bananes']*

## **Exercice 1:**

Soit une liste `a = ['1', '4', '3', '100', '20', '33', '1052', '251']`

Ecrire en une ligne une liste des éléments pairs issus de la liste `a`

## **Exercice 2:**

Ecrire un programme qui calcule la somme des éléments de la liste `a`

## Solution 1:

```
a = ['1','4','3','100','20','33','1052','251']  
[int(nb) for nb in a if int(nb) % 2 == 0]
```

## Solution 2:

```
somme = int()  
for element in a:  
    somme += int(element)  
print (somme)
```

```
sum([int(nb) for nb in a])
```



Ecrire une fonction `liste_dedup` qui :

- prend en entrée une liste
- renvoie une liste triée dont tous les éléments sont dédupliqués

Pour tester, utiliser la liste `liste_test = [1,2,2,5,5,12,4,3,2,4,1]`

```
def liste_dedup(liste_entree):  
    liste_finale = []  
    for element in liste_entree:  
        if element not in liste_finale:  
            liste_finale.append(element)  
    liste_finale.sort()  
    return liste_finale
```

On définit une liste vide de résultat (liste\_finale)

Pour chaque élément, si il n'est pas présent dans la liste, alors on append, sinon on passe à l'élément suivant

Penser à trier la liste en fin de boucle

Les dictionnaires sont des objets python (class dict).

L'objet dict est **mutable**. Il est donc possible de

- **supprimer un élément** d'un dictionnaire en utilisant la clé avec l'instruction **del**

```
dictionnaire = {'pommes': 3, 'poires' : 5, 'carottes': 6}
del dictionnaire['poires']
print (dictionnaire)
=> {'pommes': 3 , 'carottes': 6}
```

- **modifier une valeur** d'un dictionnaire en utilisant la clé correspondante et l'affectation =

```
dictionnaire['carottes'] = 15
=> {'pommes': 3, 'carottes': 15}
```

- **d'ajouter un élément** d'un dictionnaire en ajoutant une clé et l'affectation =

```
dictionnaire['tomates'] = 12
=> {'pommes': 3, 'carottes': 15, 'tomates': 12}
```

## Méthodes sur les dictionnaires

<code>dict.clear()</code>	Supprime tous les éléments du dictionnaire – il sera donc vide
<code>dict2 = dict.copy()</code>	Copie le dictionnaire dict dans dict2
<code>dict.get(key, default=None)</code>	Renvoie la valeur associée à la clé <i>key</i> ou la valeur <i>default</i> si la clé n'existe pas
<code>dict.has_key(key)</code>	Teste l'existence de la clé <i>key</i> dans le dictionnaire - Renvoie un booléen
<code>dict.items()</code>	Renvoie une liste de tuples (clé , valeur)
<code>dict.keys()</code>	Renvoie la liste de clés de dict
<code>dict.update(dict2)</code>	Ajoute les couples clés-valeurs de dict2 dans dict
<code>dict.values()</code>	Renvoie la liste de valeurs de dict



Cette liste de méthodes n'est pas exhaustive. Vous pouvez vous référer à [l'aide de python en ligne](#)

1. Créer le dictionnaire dico1 = {'pommes' : 3 , 'poires' : 5, 'carottes' : 6}
2. Affecter un dictionnaire dico2 à partir de dico1 (=)
3. Créer un nouveau dictionnaire dico3 à partir de dico1 et de la méthode copy
4. Modifier la valeur de la clé 'pommes' en la positionnant à 4
5. Afficher dico1, dico2 et dico3
6. Conclusions ?

```
dico1 = {'pommes': 3, 'poires': 5, 'carottes': 6}
dico2 = dico1
dico3 = dico1.copy()
dico1['pommes'] = 4
print('dico1 = ', dico1)
print('dico2 = ', dico2)
print('dico3 = ', dico3)
```

En sortie, on obtient:

```
dico1 = {'poires': 5, 'carottes': 6, 'pommes': 4}
dico2 = {'poires': 5, 'carottes': 6, 'pommes': 4}
dico3 = {'poires': 5, 'carottes': 6, 'pommes': 3}
```



Dico2 est en fait un alias de dico1. Toute modification sur l'un implique systématiquement une modification sur l'autre !

Ecrire une fonction *nb\_occur* qui à partir d'une chaîne, renvoie un dictionnaire contenant chaque lettre et son nombre d'occurrences.

Par exemple, si la chaîne en entrée est google, la fonction renverra {'g': 2, 'o': 2, 'e': 1, 'l': 1}

```
def nb_occur(chaine):  
    dict = {}  
    liste_mot = chaine.split(' ')  
    for mot in liste_mot:  
        for caractere in mot:  
            if caractere in dict.keys():  
                dict[caractere] += 1  
            else:  
                dict[caractere] = 1  
    return dict
```

On définit un dictionnaire de résultat vide pour commencer

On éclate la chaine en une liste de mots, sur lesquels on itérera sur chaque lettre le composant

Si le caractère est déjà présent dans les clés du dictionnaire, on incrémente de 1, sinon on initialise



# Pour aller plus loin - les modules regex et random

---



Le module regex permet d'utiliser les expressions régulières facilement sur python

```
import re
```

Méthodes pour le package re	
re.match(pattern, string)	Permet de savoir si le pattern match le string
re.findall(pattern, string)	Renvoie toutes les parties du string qui correspondent au pattern
re.search(pattern, string)	Permet de trouver si le pattern existe dans le string

Regex - Expressions	
.	N'importe quel caractère
^ / \$	Marque le début/la fin de la chaîne ou de la ligne
	Correspond au caractère ou, le regex reconnaît l'un ou l'autre
*	0, 1 ou plusieurs occurrences
+	1 ou plusieurs occurrences
?	0 ou 1 occurrence
{a,b}	Détermine un nombre d'occurrences de a à b
[[:alpha:]]	N'importe quel caractère
[[:digit:]]	N'importe quel chiffre
\d	Chiffre
\s	Caractères espace
\w	Caractère alphanumérique : lettre, chiffre ou underscore

Le module Random permet de créer des données aléatoires

```
import random
```

Méthodes du module Random	
random.seed(a)	Initialise un état aléatoire de graine 'a'
random.randrange(start, stop)	Retourne un element aléatoirement dans une liste entre start, et stop
random.randint(a,b)	Retourne un entier aléatoirement entre a et b
random.choice(liste)	Retourne un élément aléatoire dans la liste
random.shuffle(x)	Mélange la séquence x aléatoirement
random.sample(population, k)	Prend un échantillon de taille k de la population
random.random()	Retourne un float aléatoire entre 0.0 et 1.0 exclus