



# INTRODUCTION AU PYTHON

Dr. NSENGE MPIA HERITIER, PhD

# Introduction

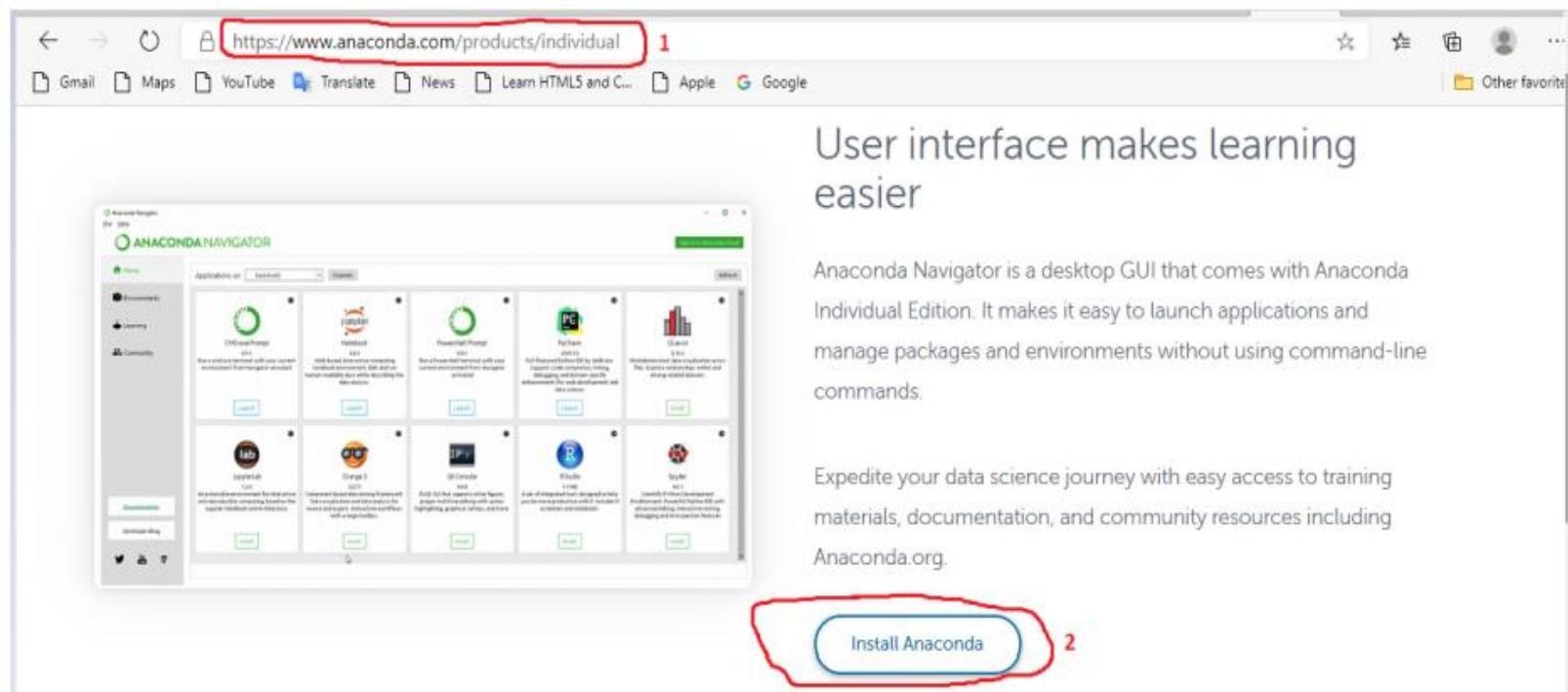
- Python présente de multiples avantages par rapport aux langages traditionnels car il est naturel (utilisation de l'indentation, typage dynamique), utilisable en ligne de commande (interactif), moderne (orienté objet), extensible (modules externes de calcul et de visualisation écrites en C/C++/fortran), utilisé pour le calcul scientifique (grâce à la réutilisation de librairies de calculs déjà existantes), libre et multi-plateforme.
- Python est un langage de programmation orienté objet interprété. Et, un programme écrit en Python n'est opérationnel que si l'interpréteur est disponible sur la machine (bien que des solutions de compilation existent). En contrepartie, il peut fonctionner dès lors que l'interpréteur est présent, quel que soit le système d'exploitation de la machine. Sous cet angle, on peut le considérer comme un langage multiplateforme.
- La *distribution Python intègre un grand nombre de librairies. Elles couvrent un large choix de domaines* (bases de données, accès réseaux, multimédia, traitements systèmes, compression, multithreading, Data science...).
- Outre les librairies standards, un grand nombre de paquetages développés par des contributeurs indépendants donne accès à des fonctionnalités spécialisées performantes. Ces paquetages nous donnent la possibilité de programmer des applications dans quasiment tous les secteurs de l'informatique

# Environnement de développement

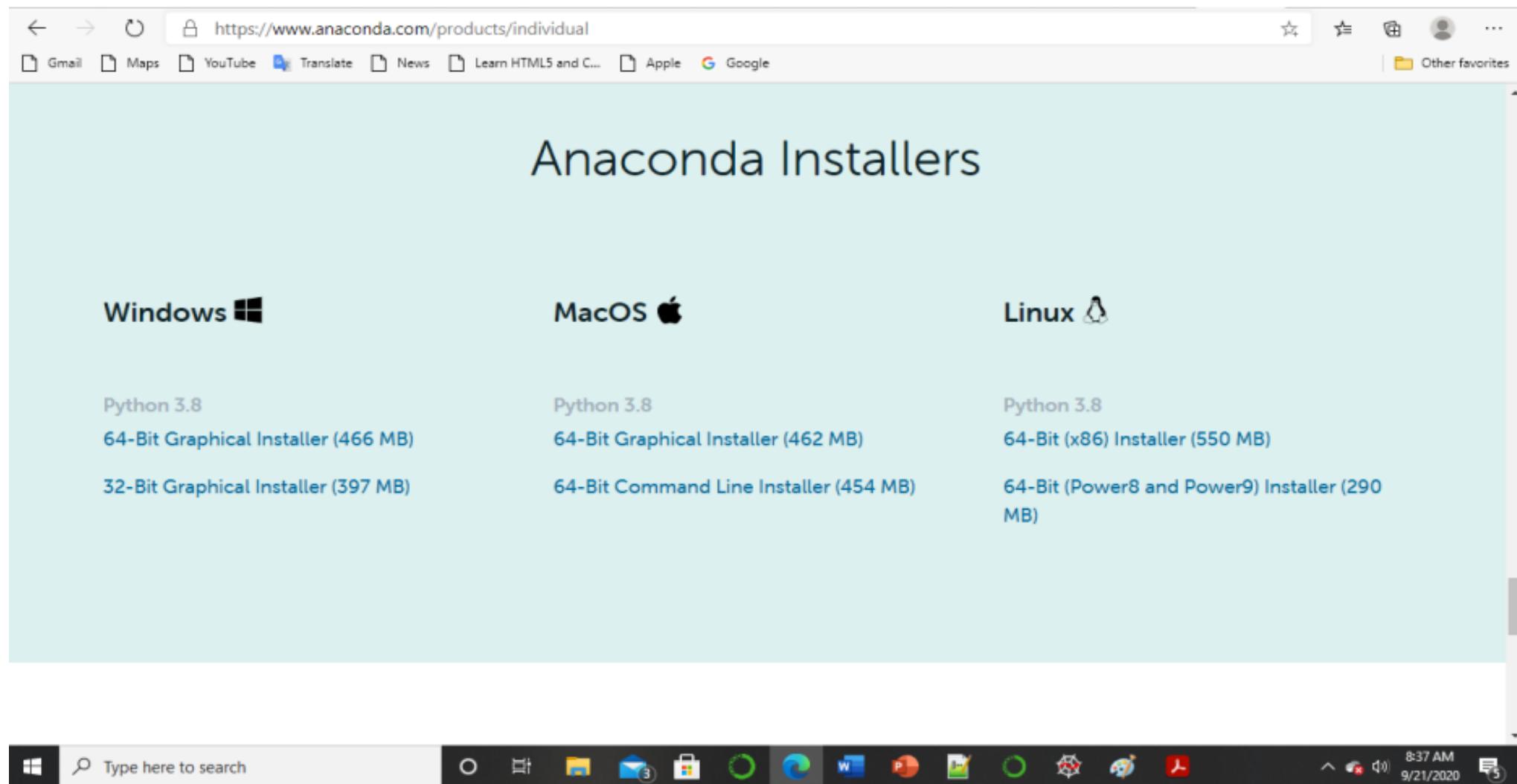
- Pour des raisons pratiques, nous utiliserons, dans ce cours, l'environnement **Anaconda**. En effet, Anaconda regroupe un ensemble d'outils gravitant autour des langages de programmation Python et R : il fournit notamment les deux environnements d'exécution. Cette distribution de Python est orientée Data Science et Machine Learning : dans ces domaines, elle est certainement la plus populaire. Anaconda s'installe aussi bien sur Windows, MacOs ou Linux.
- On peut trouver le logiciel Anaconda à partir de son site officiel : <https://www.anaconda.com>. Anaconda propose son propre gestionnaire de paquets appelé « conda ».

# Installation de Anaconda

- Premièrement, on se dirige sur le site d'Anaconda dans ce lien: <https://www.anaconda.com/products/individual> et après, on va cliquer sur Install Anaconda.



- Ce qui amènera cette interface :



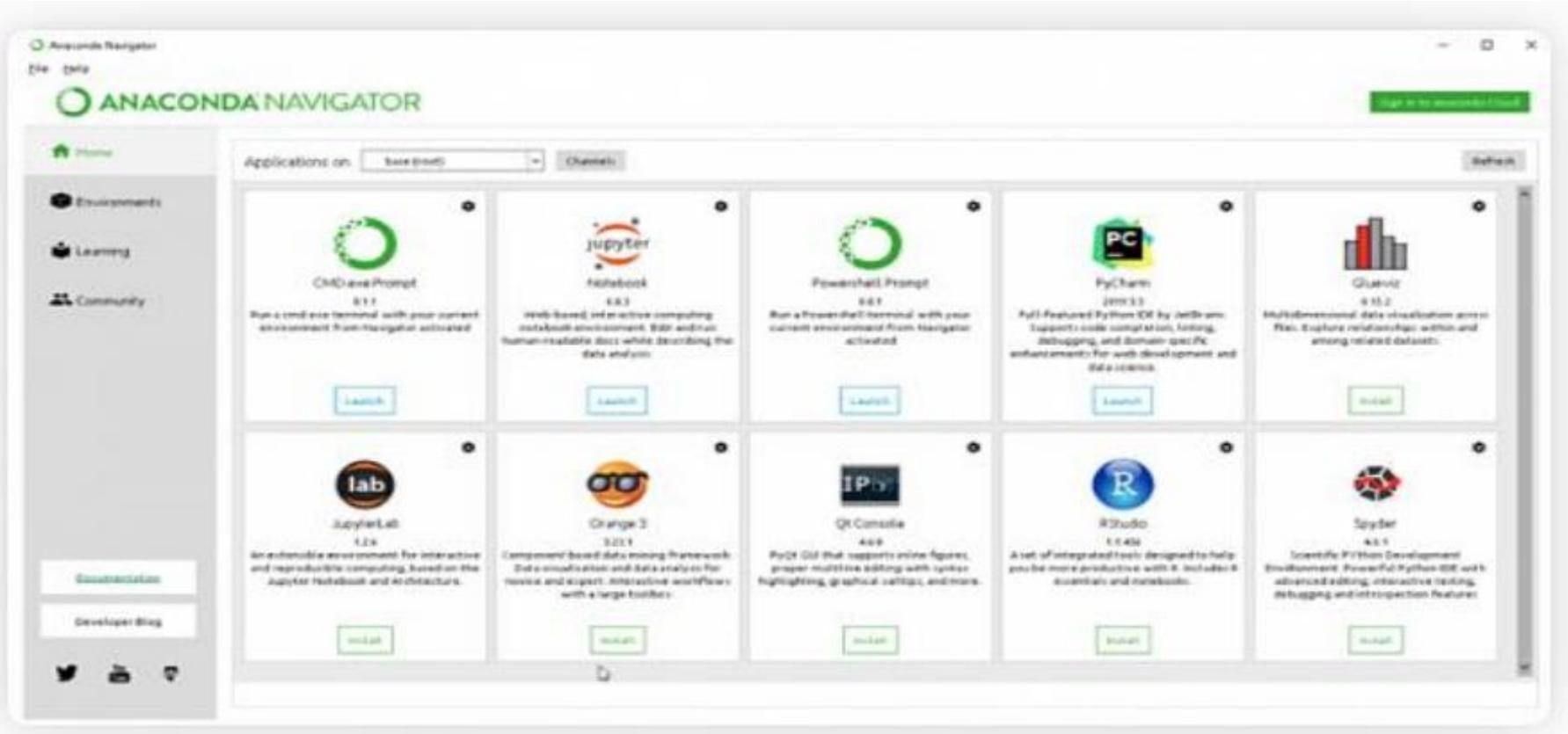
https://www.anaconda.com/products/individual

Gmail Maps YouTube Translate News Learn HTML5 and C... Apple Google Other favorites

## Anaconda Installers

Windows	MacOS	Linux
Python 3.8	Python 3.8	Python 3.8
<a href="#">64-Bit Graphical Installer (466 MB)</a>	<a href="#">64-Bit Graphical Installer (462 MB)</a>	<a href="#">64-Bit (x86) Installer (550 MB)</a>
<a href="#">32-Bit Graphical Installer (397 MB)</a>	<a href="#">64-Bit Command Line Installer (454 MB)</a>	<a href="#">64-Bit (Power8 and Power9) Installer (290 MB)</a>

- On peut ainsi télécharger une version d'Anaconda en fonction de son Système. On va installer Anaconda et après l'installation, on pourra lancer Anaconda qui se présentera comme suit :



- Dans cette interface, nous allons utiliser Jupyter et Spyder comme éditeurs pour programmer en Python

# Jupyter et Spyder

- Spyder qui signifie **S**cientific **P**ython **D**evelopment **E**nvi**R**onment, est un environnement de développement intégré gratuit (IDE) qui est inclus avec Anaconda. Il comprend des fonctions d'édition, de test interactif, de débogage et d'introspection
- Jupyter Notebook est une application web open source qu'on peut utiliser pour créer et partager des documents qui contiennent du code en direct, des équations, des visualisations et du texte. Jupyter Notebook est maintenu par les gens du Projet Jupyter. Le nom, Jupyter, vient des langages de programmation de base qu'il supporte : **Julia**, **Python** et **R**.

# Variables

- On doit rappeler qu'une variable est caractérisée par :
  - un **identificateur** : il peut contenir des lettres, des chiffres, des blancs soulignés mais il ne peut commencer par un chiffre. Minuscules et majuscules sont différenciées. Il est aussi unique.
  - un **type** : c'est une information sur le contenu de la variable qui indique à l'interpréteur python, la manière de manipuler cette information.

Etant donné que le typage est dynamique en *python*, le type n'est pas précisé explicitement, il est implicitement lié à l'information manipulée. Par exemple, en écrivant, `x = 3.4`, on ne précise pas le type de la variable `x` mais il est implicite car `x` reçoit une valeur réelle : `x` est de type **réel** ou **float** en python. Pour leur première initialisation, une variable reçoit dans la plupart des cas une constante. Les constantes sont le contraire des variables, ce sont toutes les valeurs numériques, chaînes de caractères, ..., tout ce qui n'est pas désigné par un nom. Les constantes possèdent un type mais pas d'identificateur. De ce fait, **le type de la variable sera défini par le type de la constante qui lui est affectée. Le type d'une variable peut changer, il correspond toujours au type de la dernière affectation.**

```

x = 3.5      # création d'une variable nombre réel appelée x initialisée à 3.5
              # 3.5 est un réel, la variable est de type "float"
sc = "chaîne" # création d'une variable chaîne de caractères appelée str
               # initialisée à "chaîne", sc est de type "str"
  
```

**N.B :** Pour mettre le commentaire, on utilise #

# Variables (Cont.)

On peut aussi déclarer une variable comme ceci: X=X+1 ou X+=1

## Nombres réels et entiers

Il existe deux types de nombres en python, les nombres réels *float* et les nombres entiers *int*. L'instruction x=3 crée une variable de type int initialisée à 3 tandis que y=3.0 crée une variable de type float initialisée à 3.0.

### **Exemple :**

```
x = 3
y = 3.0
print("x =", x, type(x))
print("y =", y, type(y))
```

>>>

```
x = 3 <class 'int'>
y = 3.0 <class 'float'>
```

# Variables (Cont.)

## Les opérateurs

opérateur	signification	exemple
<code>&lt;&lt; &gt;&gt;</code>	décalage à gauche, à droite	<code>x = 8 &lt;&lt; 1</code>
<code> </code>	opérateur logique <code>ou</code> bit à bit	<code>x = 8   1</code>
<code>&amp;</code>	opérateur logique <code>et</code> bit à bit	<code>x = 11 &amp; 2</code>
<code>+ -</code>	addition, soustraction	<code>x = y + z</code>
<code>+= -=</code>	addition ou soustraction puis affectation	<code>x += 3</code>
<code>* /</code>	multiplication, division	<code>x = y * z</code>
<code>//</code>	division entière, le résultat est de type réel si l'un des nombres est réel	<code>x = y // 3</code>
<code>%</code>	reste d'une division entière (modulo)	<code>x = y % 3</code>
<code>*= /=</code>	multiplication ou division puis affectation	<code>x *= 3</code>
<code>**</code>	puissance (entière ou non, racine carrée = <code>** 0.5</code> )	<code>x = y ** 3</code>

# Variables (Cont.)

Les fonctions `int` et `float` permettent de convertir un nombre quelconque ou une chaîne de caractères respectivement en un entier (arrondi) et en un nombre réel.

## Exemple

```
x = int(3.5)
y = float(3)
z = int("3")
print("x:", type(x), "    y:", type(y), "    z:", type(z))
```

>>>

```
x: <class 'int'>    y: <class 'float'>    z: <class 'int'>
```

**N.B :** Il peut arriver que la conversion en un nombre entier ne soit pas directe. Dans l'exemple qui suit, on cherche à convertir une chaîne de caractères en entier mais cette chaîne représente un réel. Il faut d'abord la convertir en réel puis en entier, c'est à ce moment que l'arrondi sera effectué.

```
y=int("3.6") #provoque une erreur
print(y)
x =int(float("3.6")) #Fonctionne
print(x)
```

# Variables (Cont.)

*Python* propose l'opérateur `//` pour les divisions entières et c'est une rare exception parmi les langages qui ne possèdent qu'un seul opérateur `/` qui retourne un entier pour une division entière excepté en *python* :

```
x = 11
y = 2
z = x // y      # Le résultat est 5 et non 5.5 car la division est entière
zz = x / y      # Le résultat est 5.5

print(z, zz)
```

## Booléen

Les booléens sont le résultat d'opérations logiques et ont deux valeurs possibles : True ou False. Voici la liste des opérateurs qui s'appliquent aux booléens:

opérateur	signification	exemple
<code>and or</code>	et, ou logique	<code>x = True or False</code> (résultat = True)
<code>not</code>	négation logique	<code>x = not x</code>

<<<

```
x = 4 < 5
print(x)      # affiche True
print(not x)  # affiche False
```

# Variables (Cont.)

## Liste des comparaisons logiques

opérateur	signification	exemple
<code>&lt; &gt;</code>	inférieur, supérieur	<code>x = 5 &lt; 5</code>
<code>&lt;= &gt;=</code>	inférieur ou égal, supérieur ou égal	<code>x = 5 &lt;= 5</code>
<code>== !=</code>	égal, différent	<code>x = 5 == 5</code>

Il existe deux autres mots-clés qui retournent un résultat de type booléen :

opérateur	signification	exemple
<code>is</code>	test d'identification	<code>"3" is str</code>
<code>in</code>	test d'appartenance	<code>3 in [3, 4, 5]</code>

# Variables (Cont.)

## Chaîne de caractères

La chaîne de caractère est comprise entre deux guillemets ou deux apostrophes, ces deux symboles sont interchangeables. Le type *python* est *str*.

```
t = "string = texte"
print(type(t), t)
t = 'string = texte, initialisation avec apostrophes'
print(type(t), t)

t = "morceau 1" \
    "morceau 2"      # second morceau ajouté au premier par l'ajout du symbole \,
# il ne doit rien y avoir après le symbole \,
# pas d'espace ni de commentaire
print(t)

t = """première ligne
seconde ligne"""\ # chaîne de caractères qui s'étend sur deux lignes
print(t)
```

>>>

```
<class 'str'> string = texte
<class 'str'> string = texte, initialisation avec apostrophes
morceau 1morceau 2
première ligne
    seconde ligne
```

**N.B :** Par défaut, le *python* ne permet pas l'insertion de caractères tels que les accents dans les chaînes de caractères, le paragraphe *par\_intro\_accent\_code* explique comment résoudre ce problème. De même, pour insérer un guillemet dans une chaîne de caractères encadrée elle-même par des guillemets, il faut le faire précédé du symbole \. La séquence \ est appelée un extra-caractère ou un caractère d'échappement.

# Variables (Cont.)

"

guillemet

'

apostrophe

\n

passage à la ligne

\\"

insertion du symbole \

\%

pourcentage, ce symbole est aussi un caractère spécial

\t

tabulation

\r

retour à la ligne, peu usité, il a surtout son importance lorsqu'on passe d'un système *Windows* à *Linux* car *Windows* l'ajoute automatiquement à tous ses fichiers textes

# Variables (Cont.)

## Formatage d'une chaîne de caractères

### Syntaxe %

Python offre une manière plus concise de former une chaîne de caractères à l'aide de plusieurs types d'informations en évitant la conversion explicite de ces informations (type **str**) et leur concaténation. Il est particulièrement intéressant pour les nombres réels qu'il est possible d'écrire en imposant un nombre de décimales fixe. Le format est le suivant :

```
".... %c1 .... %c2 " % (v1,v2)
```

Dans l'exemple ci-haut, **c1** est un code choisi parmi ceux de la table `format_print`. Il indique le format dans lequel la variable **v1** devra être transcrise. Il en est de même pour le code **c2** associé à la variable **v2**. Les codes insérés dans la chaîne de caractères seront remplacés par les variables citées entre parenthèses après le symbole **%** suivant la fin de la chaîne de caractères. Il doit y avoir autant de codes que de variables, qui peuvent aussi être des constantes.

### Exemple

```
x = 5.5
d = 7
s = "caractères"
res = "un nombre réel %f et un entier %d, une chaîne de %s, \n" \
      "un réel d'abord converti en chaîne de caractères %s" % (
          x, d, s, str(x + 4))
print(res)
res = "un nombre réel " + str(x) + " et un entier " + str(d) + \
      ", une chaîne de " + s + \
      ",\n un réel d'abord converti en chaîne de caractères " + str(x + 4)
print(res)
```

>>>

```
un nombre réel 5.500000 et un entier 7, une chaîne de caractères,
un réel d'abord converti en chaîne de caractères 9.5
un nombre réel 5.5 et un entier 7, une chaîne de caractères,
un réel d'abord converti en chaîne de caractères 9.5
```

# Variables (Cont.)

La seconde affectation de la variable **res** (dans le slide ci-dessus) propose une solution équivalente à la première en utilisant l'opérateur de concaténation **+**. Les deux solutions sont équivalentes, tout dépend des préférences de celui qui écrit le programme. La première option permet néanmoins un formatage plus précis des nombres réels en imposant par exemple un nombre défini de décimal.

Le format est le suivant :

```
"%n.df" % x
```

**n** est le nombre de chiffres total et **d** est le nombre de décimales, **f** désigne un format réel indiqué par la présence du symbole **%**.

## Exemple

```
x = 0.123456789
print(x)                  # affiche 0.123456789
print("%1.2f" % x)        # affiche 0.12
print("%06.2f" % x)        # affiche 000.12
```

# Variables (Cont.)

Il existe d'autres formats regroupés dans la table `format_print`. L'aide reste encore le meilleur réflexe car le langage *python* est susceptible d'évoluer et d'ajouter de nouveaux formats.

<code>d</code>	entier relatif
<code>e</code>	nombre réel au format exponentiel
<code>f</code>	nombre réel au format décimal
<code>g</code>	nombre réel, format décimal ou exponentiel si la puissance est trop grande ou trop petite
<code>s</code>	chaîne de caractères

## Méthode `format`

La méthode `format` propose plus d'options pour formatter le texte et son usage est de plus en plus fréquent. La méthode interprète les accolades `{}` comme des codes qu'elle remplace avec les valeurs passées en argument. Le type n'importe plus. Quelques exemples :

# Variables (Cont.)

```
print('{0}, {1}, {2}'.format('a', 'b', 'c'))      # Le format le plus simple
print('{}, {}, {}'.format('a', 'b', 'c'))        # sans numéro
print('{2}, {1}, {0}'.format('a', 'b', 'c'))      # ordre changé
print('{0}{1}{0}'.format('abra', 'cad'))          # répétition
```

>>>

```
a, b, c
a, b, c
c, b, a
abracadabra
```

## Dates format

```
import datetime
d = datetime.datetime.now()
print('{:%Y-%m-%d %H:%M:%S}'.format(d))
```

>>>

```
2020-09-14 13:47:56
```

# Les fonctions

On peut créer la fonction  $f(x)=x^2$ . Pour cela, on utilisera l'expression *lambda*.

## Exemple

```
In [9]: f=lambda x:x**2
print(f(3))
```

9

```
In [10]: f=lambda x, y:x**2 +y
print(f(3,5))
```

14

**N.B :** En général, on n'utilise pas la fonction de type lambda car elle est plus mathématique. En fait, dans notre cas, on voudra, par exemple, qu'une fonction affiche quelque chose que nous avons dans la mémoire. D'où, le mot clé *def* pour créer nos fonctions.

## Exemples

```
In [12]: def energy_potentiel(masse, hauteur, g):
    E=masse*hauteur*g
    print(f'Ep={E} Joules')
```

```
energy_potentiel(45,20, 9.8)
```

Ep=8820.0 Joules

```
In [17]: def energy_cinetique(masse, vitesse):
    E=(masse*(vitesse**2))/2
    print(f'Ec={E} Joules')
```

```
energy_cinetique(60,20)
```

Ec=12000.0 Joules

# Les fonctions (Cont.)

## N.B.:

1. Les valeurs internes des méthodes, etc. ne sont pas visibles. En fait, elles n'existent que dans le bloc où elles ont été déclarées. Elles sont ainsi locales. Et, même les arguments des méthodes sont locaux, on ne peut donc pas les imprimer.
2. Toutefois, si on veut utiliser le contenu des variables internes, on peut utiliser le mot clé *return*.

## Exemple

```
In [22]: def energy_potentiel(masse, hauteur, g):
    E=masse*hauteur*g
    return E

resultat=energy_potentiel(45,20, 9.8)
A=resultat/2
print(A)

4410.0
```

3. On peut également créer une constante comme argument d'une fonction.

## Exemple

```
In [25]: def energy_potentiel(masse, hauteur, g=9.8):
    E=masse*hauteur*g
    return E

resultat=energy_potentiel(45,20) # ou bien resultat=energy_potentiel(masse=45,hauteur=20)
A=resultat/2
print(A)

4410.0
```

# Structure de contrôle if ... else

En python, Else if est représenté par elif (Else if).

## Exemples

1. 

```
x = int(input("Please enter an integer: "))

if x>0:
    print(x,'Positif')
else:
    print(x,'Negatif')
```

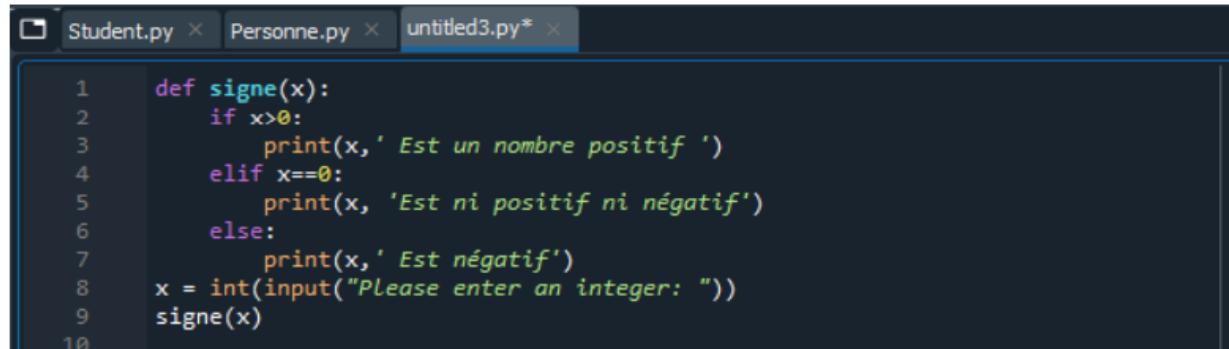
2.

```
In [26]: x = int(input("Please enter an integer: ")) #Lire ce qui est saisi au clavier

if x < 0:
    x = 0
    print("Negative changed to zero")
elif x == 0:
    print("Zero")
elif x == 1:
    print('Single')
else:
    print('More')

Please enter an integer: -6
Negative changed to zero
```

3.



```
1  def signe(x):
2      if x>0:
3          print(x, ' Est un nombre positif ')
4      elif x==0:
5          print(x, 'Est ni positif ni négatif')
6      else:
7          print(x, ' Est négatif')
8  x = int(input("Please enter an integer: "))
9  signe(x)
10
```

# Structure de contrôle For

La déclaration for en Python diffère un peu de ce à quoi vous pouvez être habitué en C ou en Pascal. Plutôt que de toujours répéter une progression arithmétique de nombres (comme en Pascal), ou de donner à l'utilisateur la possibilité de définir à la fois l'étape d'itération et la condition d'arrêt (comme en C), l'instruction for de Python itère sur les éléments de n'importe quelle séquence (une liste ou une chaîne de caractères), dans l'ordre où ils apparaissent dans la séquence.

## Exemples

```
1. words = ['Velo', 'Syasimwa', 'Python']
   for w in words:
       print(w, len(w)) # ça retourne Velo 4, Syasimwa 8, Python 6
```

Il est possible de créer une boucle for facilement avec la fonction **range** :

```
In [32]: for i in range(0,100): #On peut faire la boucle selon un pas aussi Exemple:for i in range(0,100,4) pas de 4.
          print(i)
```

# Structure de contrôle For (Exemples)

```
[2]: └ a = ['Syasimwa', 'Jéremie', 'Laure', 'Nsenge', 'Laurent']
      for i in range(len(a)):
          print(i, a[i])
```

```
0 Syasimwa
1 Jéremie
2 Laure
3 Nsenge
4 Laurent
```

Stopper une boucle avec break cas des nombres premiers

```
: 1 for i in range(2,10):
  2     if i>1:
  3         for v in range(2, int(i/2)+1):
  4             if (i%v)==0:
  5                 print(i, "n'est pas premier")
  6                 break
  7             else:
  8                 print(i, "est premier")
  9         else:
10             print(i, "n'est pas premier")
```

```
2 est premier
3 est premier
4 n'est pas premier
5 est premier
6 n'est pas premier
7 est premier
8 n'est pas premier
9 n'est pas premier
```

# Structure de contrôle While

On désire écrire 10 fois cette phrase: " *Je suis en L1 CSI/UAC* "

```
In [59]: i = 0
        while i < 10:
            print("Je suis en L1 CSI/UAC")
            i = i +1
```

# Structures des données

# Liste

La structure liste dispose de méthodes supplémentaires.

## A. list.append(x)

Ajoute un élément à la fin de la liste. Équivalent à `a[len(a):] = [x]`.

```
In [1]: ► a = ['Syasimwa', 'Jéremie', 'Laure', 'Nsenge', 'Laurent']
      b = ['Syasimwa', 'Jéremie', 'Laure', 'Nsenge', 'Laurent']
```

```
In [2]: ► x='toto'
      a[len(a):] = [x]
      a
```

```
Out[2]: ['Syasimwa', 'Jéremie', 'Laure', 'Nsenge', 'Laurent', 'toto']
```

```
In [3]: ► y='toto'
      b.append(y)
      b
```

```
Out[3]: ['Syasimwa', 'Jéremie', 'Laure', 'Nsenge', 'Laurent', 'toto']
```

# Liste (Cont.)

B. `list.extend(iterable)`. Ce qui étend la liste en y ajoutant tous les éléments de l'itérable.  
Équivalent à `a[len(a):] = iterable`.

La méthode `extend()` ajoute tous les éléments d'un itérable (liste, tuple, chaîne de caractères, etc.) à la fin de la liste.

*Un itérable est un objet Python capable de retourner ses membres un par un*, ce qui lui permet d'être parcouru par une boucle `for`.

Les exemples familiers d'itérables sont les listes, les tuples et les chaînes de caractères - toute séquence de ce type peut être parcourue par une boucle `for`.

```
▶ # creation d'une Liste
prime_numbers = [2, 3, 5]

# creation d'une autre Liste
numbers = [1, 4]

# Ajout de tous les elements de prime_numbers dans numbers
numbers.extend(prime_numbers)

print('List numbers apres extend():', numbers)

List numbers apres extend(): [1, 4, 2, 3, 5]
```

# Liste (Cont.)

## C. list.insert(*i*, *x*)

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

```
In [5]: prime_numbers = [2, 3, 5]
numbers = [1, 4]
numbers.insert(len(numbers), prime_numbers)
numbers
```

```
Out[5]: [1, 4, [2, 3, 5]]
```

```
In [6]: a=[6,7,8,9]
x=10
a.insert(0, x)
a
```

```
Out[6]: [10, 6, 7, 8, 9]
```

```
In [7]: b=[5,7,8,9]
y=6
b.insert(1, y)
b
```

```
Out[7]: [5, 6, 7, 8, 9]
```

# Liste (Cont.)

## D. list.remove(*x*)

Supprime de la liste le premier élément dont la valeur est égale à *x*. Une exception `ValueError` est levée s'il n'existe aucun élément avec cette valeur.

```
In [7]: b=[5,7,8,9]
b.remove(9)
b
```

```
Out[7]: [5, 7, 8]
```

```
In [8]: c=[5,7,8,9]
c.remove(10)
c
```

```
-----
ValueError                                                 Traceback (most recent call last)
<ipython-input-8-f1f29e29adec> in <module>
      1 c=[5,7,8,9]
----> 2 c.remove(10)
      3 c

ValueError: list.remove(x): x not in list
```

# Liste (Cont.)

## E. list.pop([*i*])

Enlève de la liste l'élément situé à la position indiquée et le renvoie en valeur de retour. Si aucune position n'est spécifiée, `a.pop()` enlève et renvoie le dernier élément de la liste (les crochets autour du *i* dans la signature de la méthode indiquent que ce paramètre est facultatif et non que vous deviez placer des crochets dans votre code. Vous retrouverez cette notation fréquemment dans le Guide de Référence de la Bibliothèque Python).

```
In [9]: ➜ my_list = [12, 'Nsenge', 'Kasereka', 14, 'Baraka', 12, 'Mpia']

#En passant l'index comme 2 pour enlever Kasereka
name = my_list.pop(2) #name contient ce qu'on a pop (Kasereka)
print(name)
print(my_list)

#méthode pop() sans index - renvoie le dernier élément
item = my_list.pop() #name contient ce qu'on a pop (Mpia)
print(item)
print(my_list)

Kasereka
[12, 'Nsenge', 14, 'Baraka', 12, 'Mpia']
Mpia
[12, 'Nsenge', 14, 'Baraka', 12]
```

# Liste (Cont.)

## G. list.index(x[, start[, end]])

Renvoie la position du premier élément de la liste dont la valeur égale *x* (en commençant à compter les positions à partir de zéro). Une exception `ValueError` est levée si aucun élément n'est trouvé.

Les parenthèses signifient que les éléments qu'elles contiennent sont facultatifs. Par exemple, si vous supprimez : "[, *start*[, *end*]]", vous aurez "`list.index(x)`" et si vous supprimez "[, *end*]", vous aurez "`list.index(x, start)`"

Les arguments optionnels *start* et *end* sont interprétés de la même manière que dans la notation des tranches et sont utilisés pour limiter la recherche à une sous-séquence particulière. L'indice renvoyé est calculé relativement au début de la séquence complète et non relativement à *start*.

```
In [27]: M list1 = [1, 2, 3, 4, 1, 1, 1, 4, 5]
# Imprime l'index de '4' (qui vaut 3. En effet, le premier 1 dans list1 a l'index 0, 2 index 1, 3 index 2 et 4 index 3.
print(list1.index(4))

list2 = ['cat', 'bat', 'mat', 'cat', 'pet']

# Imprime l'index de 'cat' dans la liste 2 qui vaut 0
print(list2.index('cat'))
liste = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
x = 8
liste.index(x, 5, 9) # 5 c'est le start et 9 représente le end (Ca veut dire que l'index de 8 vaut 7 et si je remplace
# x=8 par 1 ou 2 ou 3 ou 4 ou 5 ou 10, on aura une erreur car ces valeurs ne se trouvent pas dans la plage ]5,9])

```

3  
0

Out[27]: 7

```
In [28]: M list1 = [1, 2, 3, 4, 1, 1, 1, 4, 5]
# Imprime l'index de '4' (qui vaut 3. En effet, le premier 1 dans list1 a l'index 0, 2 index 1, 3 index 2 et 4 index 3.
print(list1.index(4))

list2 = ['cat', 'bat', 'mat', 'cat', 'pet']

# Imprime l'index de 'cat' dans la liste 2 qui vaut 0
print(list2.index('cat'))
liste = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
x = 10
liste.index(x, 5, 9)
```

3  
0

---

```
-----  
ValueError                                                 Traceback (most recent call last)
<ipython-input-28-4aa94cf43c90> in <module>
      10 liste = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
      11 x = 10
--> 12 liste.index(x, 5, 9)

ValueError: 10 is not in list
```

# Liste (Cont.)

## H. list.count(x)

Renvoie le nombre d'éléments ayant la valeur  $x$  dans la liste.

```
# creation de la liste
numbers = [2, 3, 5, 2, 11, 2, 7]
# vérifier le nombre de 2 dans la liste
count = numbers.count(2)
print('Nombre de fois que le nombre 2 revient dans la liste est ', count)
```

Nombre de fois que le nombre 2 revient dans la liste est 3

# Liste (Cont.)

## I. list.sort(key=None, reverse=False)

Ordonne les éléments dans la liste (les arguments peuvent personnaliser l'ordonnancement, voir `sorted()` pour leur explication).

```
In [30]: numbers = [6, 9, 3, 1]
numbers_sorted = sorted(numbers)
numbers_sorted
```

```
Out[30]: [1, 3, 6, 9]
```

```
In [31]: numbers = [6, 9, 3, 1]
numbers_sorted = sorted(numbers, key=None, reverse=False)
numbers_sorted
```

```
Out[31]: [1, 3, 6, 9]
```

```
In [32]: numbers = [6, 9, 3, 1]
numbers_sorted = sorted(numbers, key=None, reverse=True)
numbers_sorted
```

```
Out[32]: [9, 6, 3, 1]
```

```
In [36]: numbers = [6, 9, 3, 1]
numbers_sorted = sorted(numbers, reverse=True) #Pour les listes ne possédant pas les string comme valeur, Key n'est pas mis
numbers_sorted
```

```
Out[36]: [9, 6, 3, 1]
```

```
In [38]: words = ['banana', 'pie', 'Washington', 'book']
sorted(words, key=len, reverse=True) #Affiche en ordre de décroissant de la longueur de chaque valeur(key=len, reverse=True)
```

```
Out[38]: ['Washington', 'banana', 'book', 'pie']
```

```
In [39]: words = ['banana', 'pie', 'Washington', 'book']
sorted(words, key=len, reverse=False) #Affiche en ordre de croissant de la longueur de chaque valeur (key=len, reverse=False)
```

```
Out[39]: ['pie', 'book', 'banana', 'Washington']
```

# Liste (Cont.)

## J. list.reverse()

Inverse l'ordre des éléments dans la liste.

```
▶ # creation de la liste
prime_numbers = [2, 3, 5, 7]
# reverse des éléments de la liste
prime_numbers.reverse()
print('Liste renversée:', prime_numbers)
```

Liste renversée: [7, 5, 3, 2]

## K. list.copy()

Renvoie une copie superficielle de la liste.

```
▶ prime_numbers = [2, 3, 5]
numbers = prime_numbers.copy()
print('Liste copiée:', numbers)
```

Liste copiée: [2, 3, 5]

# Tuple

C'est une liste qu'on ne peut pas modifier, car elle est protégée et utilise moins de mémoire. Un tuple se définit comme une liste, mais on utilise comme délimiteur des parenthèses au lieu des crochets. À la différence des listes, les tuples, une fois créés, ne peuvent être modifiés : on ne peut plus y ajouter d'objet ou en retirer.

## Exemple

```
tuple_vide = ()  
tuple_non_vide = (1,) # est équivalent à ci-dessous  
tuple_non_vide = 1,  
tuple_avec_plusieurs_valeurs = (1, 2, 3, 4, 10, 50)
```

Bref, ces deux structures (list, tuple), avec la structure *string*, sont appelées des séquences contenant des éléments selon un ordre, ainsi elles ont des indices permettant d'accéder aux éléments selon leurs indices.

# Indexing

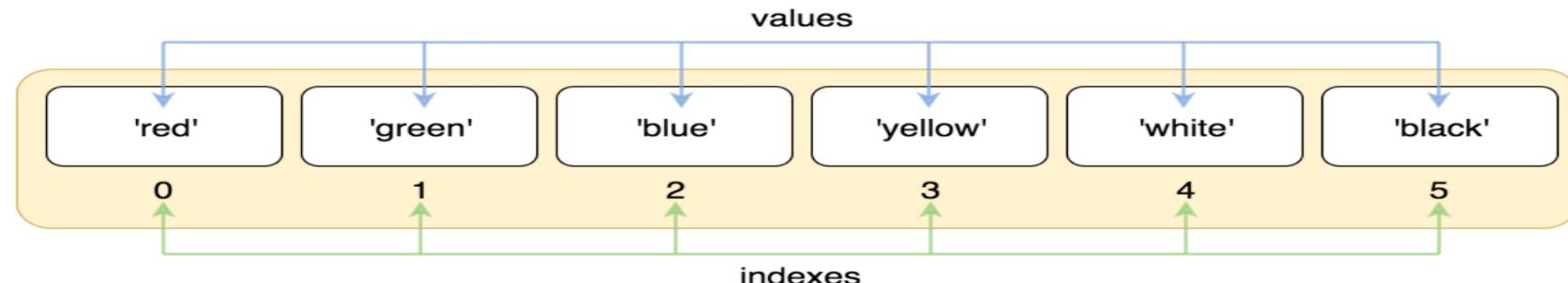
- L'indexing n'est pas une structure des données, mais plutôt la technique d'accès aux éléments (des listes) via leurs index.

Exemple:

```
couleurs = ['red', 'green', 'blue', 'yellow', 'white', 'black']
```

Ici, nous avons défini une liste de couleurs. Chaque élément de la liste possède une valeur (*le nom de la couleur*) et un index (*sa position dans la liste*).

Python utilise une indexation basée sur zéro. Cela signifie que le premier élément(valeur 'red') a un index 0, le second(valeur 'green') a un index 1, et ainsi de suite:



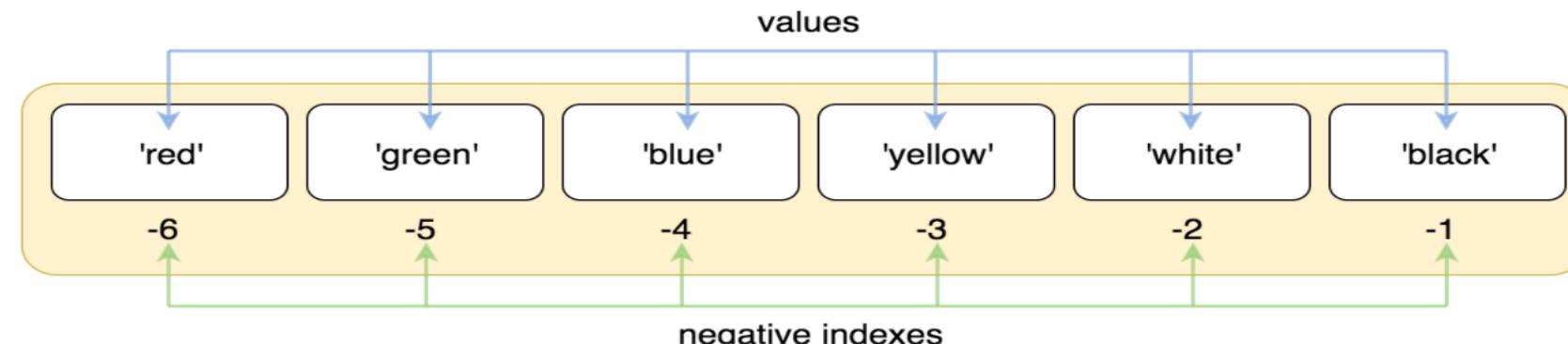
```
► villes=['Kinshasa', 'Kisangani','Butembo','Bunia','Isiro']
print(villes[1]) #Renvoie ['Kisangani']
print(villes[0]) #Renvoie ['Kinshasa']
print(villes[4]) #Renvoie ['Isiro']
```

```
Kisangani
Kinshasa
Isiro
```

# Indexing (Cont.)

## Les index négatifs

- En utilisant l'indexation, nous pouvons facilement obtenir n'importe quel élément par sa position. C'est pratique si nous utilisons la position de la tête d'une liste.
- Mais qu'en est-il si nous voulons prendre le dernier élément d'une liste ? Ou l'avant-dernier élément ?
- Dans ce cas, nous voulons énumérer les éléments de la queue d'une liste. Pour répondre à cette exigence, il existe **l'indexation négative**. Ainsi, au lieu d'utiliser des index de zéro et plus, on peut utiliser des index de -1 et moins:



# Indexing (Cont.)

- Dans le système d'indexation négative, -1 correspond au dernier élément de la liste (valeur 'black'), -2 à l'avant-dernier (valeur 'white'), et ainsi de suite.

```
▶ colors = ['red', 'green', 'blue', 'yellow', 'white', 'black']
print(colors[-1]) #Renvoie ['black']
print(colors[-2])
print(colors[-6])
```

black  
white  
red

- Auparavant, nous utilisions l'indexation uniquement pour accéder au contenu d'une cellule de liste. Mais il est également possible de modifier le contenu d'une cellule en utilisant une opération d'affectation :

```
▶ basket = ['bread', 'butter', 'milk']
basket[0] = 'cake' #On a affecté en remplaçant bread par cake
basket[-1] = 'water' #On a affecté water en fin pour remplacer milk
basket
```

14]: ['cake', 'butter', 'water']

# Indexing (Cont.)

- Nous pouvons également supprimer facilement tout élément de la liste en utilisant l'indexation et l'instruction ***del*** :

```
[15]: ► basket = ['bread', 'butter', 'milk']
      del basket[0] #On supprime bread
      print(basket) #On ne reste qu'avec butter et milk ['butter','milk']. Butter a maintenant l'index 0
      del basket[1] #On supprime milk
      print(basket)

      ['butter', 'milk']
      ['butter']
```

- *Les opérations d'indexation en lecture seule fonctionnent parfaitement bien pour tous les types séquentiels* (liste, dictionnaire, dataset et classes définies par l'utilisateur). Mais les opérations d'affectation et de suppression ne sont pas applicables aux **types séquentiels immuables** (string, tuple, int, bool, range, decimal...)

# Slicing

- Comme Indexing, le slicing *est une technique d'accès à une fourchette d'éléments en indiquant la fourchette.*
- *Comme nous l'avons montré, l'indexation permet d'accéder à une seule cellule d'une liste, de la modifier ou de la supprimer.*
- *Le slicing est utile quand nous voulons obtenir une sous-liste de la liste.*
- *Ou si nous voulons mettre à jour un groupe de cellules en une seule fois*
- *Ou si l'on veut se lancer dans une frénésie et étendre une liste avec un nombre arbitraire de nouvelles cellules dans n'importe quelle position.*

# Usage basic de Slicing

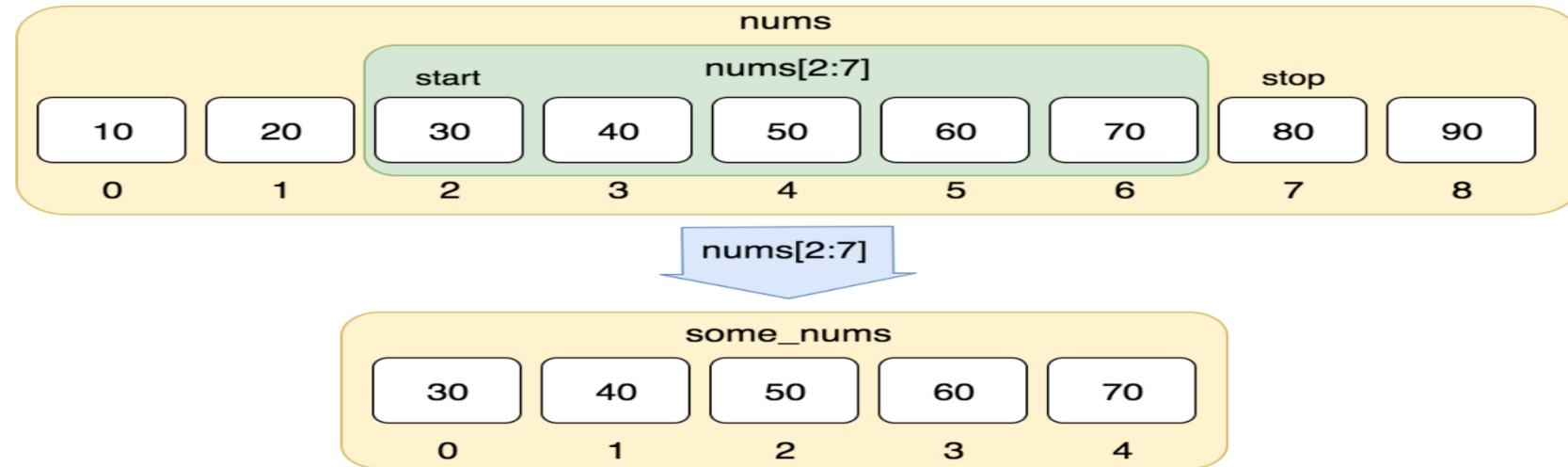
- nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
- *Que faire si l'on veut prendre une sous-liste de la liste nums ?*

```
▶ nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
some_nums = nums[2:7]
some_nums

L6]: [30, 40, 50, 60, 70]
```

- Voici donc notre premier exemple de tranche (*slicing*) : 2:7.
- La syntaxe complète d'une tranche est la suivante :
  - *start:stop:step*.
- *start* fait référence à l'indice de l'élément qui est utilisé comme début de notre tranche.
- *stop* fait référence à l'indice de l'élément que nous devons arrêter juste avant pour terminer notre tranche.
- *step* vous permet de prendre chaque nième élément dans une plage *start:stop*.
- Dans notre exemple, *start* est égal à 2, donc notre tranche commence à la valeur 30. *stop* est égal à 7, donc le dernier élément de la tranche est 70 avec l'index 6.
- A la fin, slice crée une nouvelle liste (nous l'avons nommée *some\_nums*) avec les éléments sélectionnés.

# Usage basic de Slicing (Cont.)



Nous n'avons pas utilisé de pas (step) dans notre tranche, donc nous n'avons sauté aucun élément et obtenu toutes les valeurs dans l'intervalle  
 Dans le slide qui vient, nous allons illustrer un exemple avec les steps

# Usage basic de Slicing avec le step

```
In [21]: nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
some_nums = nums[1:4:2] #On commence par la valeur ayant l'index 1 (20) on va jusqu'au quatrième élément (40) en faisant
#le pas de 2. cad on saute 30 on va dans 40
some_nums
```

Out[21]: [20, 40]

```
In [24]: nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
some_nums = nums[1:6:2] #On commence par la valeur ayant l'index 1 (20) on va jusqu'au sixième élément (60) en faisant
#le pas de 2. Si on ne précise pas de pas c'est donc après chaque élément on passe au suivant. cad on saute 30 et on saute
#50 on va dans 40
some_nums
```

Out[24]: [20, 40, 60]

```
In [25]: villes=['Kinshasa', 'Kisangani', 'Butembo', 'Bunia', 'Isiro', 'Beni', 'Boma', 'Goma', 'Boyoma']
print(villes[1:6:2]) #On commence par la valeur ayant l'index 1 (Kisangani) on va jusqu'au sixième élément (Beni)
#en faisant le pas de 2. cad on saute Butembo on arrive à Bunia et on saute Isiro on arrive à Beni
['Kisangani', 'Bunia', 'Beni']
```

```
In [27]: villes=['Kinshasa', 'Kisangani', 'Butembo', 'Bunia', 'Isiro', 'Beni', 'Boma', 'Goma', 'Boyoma']
print(villes[2:7:3]) #On commence par la valeur ayant l'index 2 (Butembo) on va jusqu'au septième élément (Boma)
#en faisant le pas de 3. cad on saute Bunia et Isiro on arrive à Beni mais comme le dernier élément est Boma, et qu'en faisant
#le pas de 3 encore on sera hors la plage, le programme ne retourne que ['Butembo', 'Beni']
['Butembo', 'Beni']
```

# Prendre chaque **nième** élément d'une liste

- Que faire *si l'on veut avoir seulement tous les deuxièmes éléments* de **nums** ? C'est là que le paramètre **step** entre totalement en jeu :

```
In [28]: nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
nums[::2] #avant Le premier : on est met rien, ca veut dire commencer par le debut de la liste est entre les deux :: on
#devrait mettre la valeur de stop. Mais comme on considere les elements de toute la liste en faisant le step de 2, on
#n'y a pas aussi mis une valeur. On voit qu'apres 10, il a saute a 30, il a saute a 50, a 70 et a 90 car il y a le step de
#2. Ceci peut s'ecrire aussi nums[0::2] ou nums[0:9:2] tel que 9 le dernier element de la liste
```

```
Out[28]: [10, 30, 50, 70, 90]
```

```
In [33]: nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
nums[3::2] #Ici en donnant la valeur de start et de step, nous avons commence par l'element ayant l'index 3 (40)
```

```
Out[33]: [40, 60, 80]
```

- Et si nous ne voulons pas inclure certains éléments à la fin, nous pouvons également ajouter le paramètre **stop** :

```
In [34]: nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
nums[1:-3:2] #On a enleve les trois derniers elements de la liste (90,80,70) et commence au deuxième element (20) en faisant
#le pas de 2.
```

```
Out[34]: [20, 40, 60]
```

# Utilisation de l'étape négative et de la liste inversée

- Nous pouvons utiliser un pas négatif pour obtenir une liste inversée:

```
In [35]: nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
nums[::-1]
```

```
Out[35]: [90, 80, 70, 60, 50, 40, 30, 20, 10]
```

- Le **pas negatif change à une certaine manière, le fonctionnement de la notation de la tranche.**
  - Il fait que la tranche est construite à partir de la queue de la liste.
  - Donc, elle va du *dernier élément au premier élément*.
  - C'est pourquoi nous obtenons une liste inversée avec un pas négatif.
- En raison de cette particularité, le **début** et la **fin** doivent également être indiqués de droite à gauche. Par exemple, si vous voulez avoir une liste inversée qui commence à 80 :

```
In [39]: nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
nums[-2::-1] # -2 represente le start (80) qui devient notre deuxième element car -1 a inverse la liste
#et entre :: on mettrait la valeur du top. Comme rien n'a été mis, cela veut dire qu'on prend tout
```

```
Out[39]: [80, 70, 60, 50, 40, 30, 20, 10]
```

```
In [40]: nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
nums[-2:3:-1] # 3 stop la liste trois éléments après le premier dans un ordre renversé
```

```
Out[40]: [80, 70, 60, 50]
```

# Utilisation de l'étape négative et de la liste inversée

- Nous pouvons utiliser la valeur stop pour arrêter la prise avant un certain élément. Par exemple, n'incluons pas les valeurs 20 et 10 :

```
In [71]: ┆ nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
      nums[-2:1:-1] #Nous avons utilisé 1 pour l'indice d'arrêt (stop), qui est l'élément avec la valeur 20.
      #Ainsi, nous allons de 80 à 30, sans inclure la valeur 20.

Out[71]: [80, 70, 60, 50, 40, 30]
```

# Couper (Slice) et copier

- Une chose importante à noter - est que la tranche de liste crée une copie superficielle de la liste initiale. Cela signifie que nous pouvons modifier la nouvelle liste en toute sécurité et que cela n'affectera pas la liste initiale :

```
In [77]: nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
        premier_cinq_elements = nums[0:5] #Retourne [10, 20, 30, 40, 50]
        premier_cinq_elements[2]=3 #On modifie la valeur de l'index 2(c'est 30) de la nouvelle liste, cad 30 on le remplace par 3
        print(premier_cinq_elements)
        print(nums)

[10, 20, 3, 40, 50]
[10, 20, 30, 40, 50, 60, 70, 80, 90]
```

- Bien que nous ayons muté l'élément sous l'indice 2, cela n'affecte pas la liste nums, car la liste premier\_cinq\_elements est une copie partielle de la liste nums.
- Il existe la forme la plus courte de la notation des tranches - juste des deux points nums[ :]

```
In [80]: nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
        nums_copy = nums[:] #[:prend toutes les données de nums [10, 20, 30, 40, 50, 60, 70, 80, 90] et on les copie dans nums_copy
        nums_copy[0] = 33 #on remplace 10 qui a l'indice 0 par 33
        print(nums_copy)
        print(nums)

[33, 20, 30, 40, 50, 60, 70, 80, 90]
[10, 20, 30, 40, 50, 60, 70, 80, 90]
```

# Création d'un objet tranche (slice)

- Il peut arriver que nous voulons utiliser la même tranche à plusieurs reprises.
- Sur ce, on crée un objet tranche au lieu d'utiliser uniquement la forme syntaxique de slice, ou utilise la fonction `slice()`:

```
In [87]: cinq_elements_apres_le_second = slice(2, 2 + 5)# retourne slice(2, 7, None)
nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
colors = ['red', 'green', 'blue', 'yellow', 'white', 'black', 'silver']
nums[cinq_elements_apres_le_second] #C'est comme si on faisait nums[2:7] car cinq_elements_apres_le_second contient
#slice(2, 7, None) 2 comme start et 7 comme stop None c'est que le step n'est pas connu. Ceci retourner donc cette liste
#[30, 40, 50, 60, 70]
colors[cinq_elements_apres_le_second]
```

Out[87]: ['blue', 'yellow', 'white', 'black', 'silver']

- La fonction `slice()` accepte les arguments dans le même ordre que dans la notation `slice`, et si vous devez sauter un élément, utilisez simplement `None` :

```
In [91]: nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
tout_sauf_les_deux_derniers = slice(None, -2) #retourne slice(None, -2, None) c'est comme nums[:-2]
nums[tout_sauf_les_deux_derniers] #Retourne [10, 20, 30, 40, 50, 60, 70]
inverser = slice(None, None, -1) #Retourne slice(None, None, -1) c'est comme nums[::-1]
nums[inverser] #Inverse tous les éléments [90, 80, 70, 60, 50, 40, 30, 20, 10]
```

Out[91]: [90, 80, 70, 60, 50, 40, 30, 20, 10]

# Affectation des tranches

- Python supporte l'opération d'assignation de tranche, qui nous permet de faire un tas d'opérations soignées sur une liste existante. Contrairement aux opérations de tranche précédentes, celles-ci modifient l'objet original sur place.
- C'est pourquoi elles ne sont pas applicables aux types séquentiels immuables.

**Remplacer une partie d'une liste:** L'affectation de tranches vous permet de mettre à jour une partie d'une liste avec de nouvelles valeurs :

```
In [94]: nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
nums[:4] = [1,2,3,4] #nums[:4] retourne une sous liste de 4 premiers elements [10,20,30,40] et nums[:4]=[1,2,3,4] veut
#dire qu'on remplace cette tranche de quatre premiers elements par [1,2,3,4]. Ce qui fait que notre liste nums devient
#[1, 2, 3, 4, 50, 60, 70, 80, 90]
nums

Out[94]: [1, 2, 3, 4, 50, 60, 70, 80, 90]
```

**Remplacer et redimensionner une partie de la liste:** Nous pouvons remplacer une partie d'une liste par un plus gros morceau à la place :

```
In [95]: nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
nums[:4] = [1,2,3,4,5,6,7] #On recupere La sous Liste de 4 premiers elements et cette sous Liste on y ajoute 3 autres
#elements [5,6,7] et si nous verifions le contenu de nums, on aura [1, 2, 3, 4, 5, 6, 7, 50, 60, 70, 80, 90]
nums

Out[95]: [1, 2, 3, 4, 5, 6, 7, 50, 60, 70, 80, 90]
```

# Affectation des tranches (Cont.)

- Dans le cas de l'exemple ci-haut, nous nous avons étendu la liste originale.
- Il est également possible de remplacer un gros morceau par un plus petit nombre d'éléments :

```
In [96]: nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
nums[:4] = [1] #On remplace Les quatres premiers elements par 1
nums
```

Out[96]: [1, 50, 60, 70, 80, 90]

**Remplacer chaque n-ième élément:** L'ajout d'un pas permet de remplacer chaque n-ième élément par une nouvelle valeur :

```
In [97]: nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
nums[::2] = [1,1,1,1,1]#On remplace Les valeurs apres 2 pas
nums
```

Out[97]: [1, 20, 1, 40, 1, 60, 1, 80, 1]

- L'utilisation de l'affectation par tranche avec étape fixe une limite à la liste que nous fournissons pour l'affectation. La liste fournie doit correspondre exactement au nombre d'éléments à remplacer. Si la longueur ne correspond pas, Python lève l'exception :

```
In [98]: nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
nums[::2] = [1,1,1] #Il y aura une erreur car en faisant les pas de 2 on doit passer par 10, 30, 50, 70 et 90, cad cinq
#elements mais ici on affecte une liste [1,1,1] qui n'a que 3 elements au Lieu de 5

-----
ValueError                                                 Traceback (most recent call last)
<ipython-input-98-4310f3e2dce1> in <module>
      1 nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
----> 2 nums[::2] = [1,1,1] #Il y aura une erreur car en faisant les pas de 2 on doit passer par 10, 30, 50, 70 et 90, cad ci
      nq
      3 #elements mais ici on affecte une liste [1,1,1] qui n'a que 3 elements au lieu de 5

ValueError: attempt to assign sequence of size 3 to extended slice of size 5
```

# Suppression de tranches

- Nous pouvons également utiliser l'instruction ***del*** pour supprimer une tranche d'une liste :

```
In [100]: nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
          del nums[3:7] #On supprime de L'element de L'index 3 (40) jusqu'au septième element (70).
          #Ici, nous avons supprimé un tas d'éléments au milieu de la liste nums.
          nums

Out[100]: [10, 20, 30, 80, 90]
```

- Nous pouvons également fournir un paramètre de pas pour découper et supprimer chaque n-ième élément :

```
In [101]: nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
          del nums[::2] #On supprime Le premier element de nums, on saute on va supprimer 30, on saute on supprime 50, on saute
          #on supprime 70 et on saute on supprime 90
          nums

Out[101]: [20, 40, 60, 80]
```

- Avec la syntaxe complète, nous pouvons définir des limites pour les éléments à supprimer :

```
In [102]: nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
          del nums[1:7:2] #On supprime en commençant par 20 qui est de L'indice 1 et on se limite au septième element en faisant
          #Les pas de 2. Ce qui fait que 70 et 90 reste car Le stop est 7.
          nums

Out[102]: [10, 30, 50, 70, 80, 90]
```

# Dictionnaire

- C'est une structure des données comme liste, tuple, ...
- Le dictionnaire en Python est une collection non ordonnée de valeurs de données, utilisée pour stocker des valeurs de données comme une carte, qui, contrairement aux autres types de données qui ne contiennent qu'une seule valeur en tant qu'élément, le dictionnaire contient une paire **clé:valeur**.
- La clé-valeur est fournie dans le dictionnaire pour le rendre plus optimisé.
- En Python, un dictionnaire peut être créé en plaçant une séquence d'éléments entre accolades {}, séparés par une virgule.
- Le dictionnaire contient des paires de valeurs, l'une d'entre elles étant la clé et l'autre l'élément correspondant à la paire clé:valeur.
- Les valeurs d'un dictionnaire peuvent être de n'importe quel type de données et peuvent être dupliquées, tandis que **les clés ne peuvent pas être répétées et doivent être immuables (cad pas de modification, suppression...)**.

# Création d'un dictionnaire

```
In [41]: ➤ # Création d'un dictionnaire avec des clés en nombre entiers
Dict = {1: 'Geeks', 2: 'For', 3: 'Geeks'} #Cles ici sont 1, 2, 3
print("\nDictionnaire avec des clés en nombre entiers: ")
print(Dict)

# Création d'un dictionnaire avec des clés mixtes
Dict = {'Name': 'Geeks', 1: [1, 2, 3, 4]} #Cles ici sont Name et 1
print("\nDictionnaire en utilisant des clés miste: ")
print(Dict)
```

```
Dictionnaire avec des clés en nombre entiers:
{1: 'Geeks', 2: 'For', 3: 'Geeks'}

Dictionnaire en utilisant des clés miste:
{'Name': 'Geeks', 1: [1, 2, 3, 4]}
```

```
In [46]: ➤ ## En bas nous illustrons trois autres façons de créer un dictionnaire
#Creation d'un dictionnaire vide
Dict1 = {}
print("Dictionnaire vide: ")
print(Dict1)

# Creation d'un dictionnaire en utilisant la méthode dict() de python
Dict2 = dict({1: 'Geeks', 2: 'For', 3:'Geeks'})
print("\nDictionnaire en utilisant la méthode dict() de python: ")
print(Dict2)

# Crédit d'un dictionnaire avec chaque élément en tant que Pair (seulement deux valeurs ex. (1, 'Geeks'), (2, 'For'))
Dict3 = dict([(1, 'Geeks'), (2, 'For')])
print("\nDictionnaire avec chaque élément en tant que Pair: ")
print(Dict3)
```

```
Dictionnaire vide:
{}

Dictionnaire en utilisant la méthode dict() de python:
{1: 'Geeks', 2: 'For', 3: 'Geeks'}

Dictionnaire avec chaque élément en tant que Pair:
{1: 'Geeks', 2: 'For'}
```

# Dictionnaire imbriqué

- En Python, un dictionnaire imbriqué est un dictionnaire à l'intérieur d'un dictionnaire. Il s'agit d'une collection de dictionnaires en un seul dictionnaire.

```
nested_dict = { 'dictA': { 'key_1': 'value_1'},  
                 'dictB': { 'key_2': 'value_2'}}
```

```
>>> people = {1: { 'name': 'John', 'age': '27', 'sexe': 'Masculin'},  
             2: { 'name': 'Marie', 'age': '22', 'sexe': 'Feminin'}}  
  
print(people)  
  
{1: { 'name': 'John', 'age': '27', 'sexe': 'Masculin'}, 2: { 'name': 'Marie', 'age': '22', 'sexe': 'Feminin'}}
```

# Accéder aux éléments d'un dictionnaire imbriqué

```
In [49]: ➜ people = {1: {'name': 'John', 'age': '27', 'sexe': 'Masculin'},  
                2: {'name': 'Marie', 'age': '22', 'sexe': 'Feminin'}}  
  
print(people[1]['name'])  
print(people[1]['age'])  
print(people[1]['sexe'])
```

John

27

Masculin

# Ajouter un élément dans un dictionnaire imbriqué

- Nous avons ajouté un element (Married) dans le nouveau dictionnaire numero 3.

```
In [51]: ➜ people = {1: {'name': 'John', 'age': '27', 'sexe': 'Masculin'},  
                2: {'name': 'Marie', 'age': '22', 'sexe': 'Feminin'}}  
  
people[3] = {}  
  
people[3]['name'] = 'Luna'  
people[3]['age'] = '24'  
people[3]['sexe'] = 'Female'  
people[3]['married'] = 'No'  
print(people[3])  
  
{'name': 'Luna', 'age': '24', 'sexe': 'Female', 'married': 'No'}
```

```
In [52]: ➜ print(people)  
  
{1: {'name': 'John', 'age': '27', 'sexe': 'Masculin'}, 2: {'name': 'Marie', 'age': '22', 'sexe': 'Feminin'}, 3: {'name': 'Lu  
na', 'age': '24', 'sexe': 'Female', 'married': 'No'}}
```

## Supprimer des éléments d'un dictionnaire imbriqué

```
In [54]: ➜ people = {1: {'name': 'John', 'age': '27', 'sex': 'Male'},  
 2: {'name': 'Marie', 'age': '22', 'sex': 'Female'},  
 3: {'name': 'Luna', 'age': '24', 'sex': 'Female', 'married': 'No'},  
 4: {'name': 'Peter', 'age': '29', 'sex': 'Male', 'married': 'Yes'}}  
  
del people[3]['married']  
del people[4]['married']  
  
print(people[3])  
print(people[4])  
  
{'name': 'Luna', 'age': '24', 'sex': 'Female'}  
{'name': 'Peter', 'age': '29', 'sex': 'Male'}
```

## Supprimer le dictionnaire d'un dictionnaire imbriqué

```
In [55]: ➜ people = {1: {'name': 'John', 'age': '27', 'sex': 'Male'},  
 2: {'name': 'Marie', 'age': '22', 'sex': 'Female'},  
 3: {'name': 'Luna', 'age': '24', 'sex': 'Female'},  
 4: {'name': 'Peter', 'age': '29', 'sex': 'Male'}}  
  
del people[3], people[4]  
print(people)  
  
{1: {'name': 'John', 'age': '27', 'sex': 'Male'}, 2: {'name': 'Marie', 'age': '22', 'sex': 'Female'}}
```

# Itérer dans un dictionnaire imbriqué

```
In [56]: ➜ people = {1: {'Name': 'John', 'Age': '27', 'Sex': 'Male'},  
                2: {'Name': 'Marie', 'Age': '22', 'Sex': 'Female'}}  
  
for p_id, p_info in people.items():  
    print("\nPerson ID:", p_id)  
  
    for key in p_info:  
        print(key + ':', p_info[key])
```

Person ID: 1

Name: John

Age: 27

Sex: Male

Person ID: 2

Name: Marie

Age: 22

Sex: Female

# Points clés à retenir pour les dictionnaires imbriqués

- Un dictionnaire imbriqué est une collection non ordonnée de dictionnaires.
- Le découpage en tranches (**slicing**) d'un dictionnaire imbriqué n'est pas possible.
- Nous pouvons réduire ou augmenter le nombre de dictionnaires imbriqués selon les besoins.
- Comme les dictionnaires, ils ont aussi une clé et une valeur.
- Les dictionnaires sont accessibles à l'aide de la clé.

# Listes Compréhension

On peut avoir une liste vide et on veut écrire tous les carrés des nombres de 0 à 9. On peut pour cela avoir un tel algorithme :

```
liste_Carre=[]
for i in range(10):
    liste_Carre.append(i**2)
print(liste_Carre)
```

La liste comprehension consiste en fait d'insérer la boucle *for* à l'intérieur d'une liste. Ce qui fait que le code ci-dessus peut s'écrire comme suit :

```
liste_Carre = [ i**2 for i in range(10) ]
print(liste_Carre)
```

Ce qui donnerait le même résultat. En fait, il existe trois raisons d'utilisation de liste comprehension :

1. On réduit le nombre de ligne d'écriture de code
2. C'est plus professionnel dans le monde Python
3. Le processus de création à l'intérieur se fait plus rapidement qu'avec la méthode classique

# Exemple d'illustration d'exécution en termes de performance d'une liste compréhension

Exemple de résolution d'un problème sans utiliser liste comprehension

```
In [57]: ➜ import time
        debut = time.time()
        listeTime=[]
        for i in range(3000000) :
            listeTime.append(i**2)

        fin= time.time()
        print(fin - debut)

2.383528470993042
```

Exemple de résolution d'un problème en utilisant liste comprehension

```
In [58]: ➜ import time
        debut = time.time()
        listeTime=[i**2 for i in range(3000000)]
        fin= time.time()
        print(fin - debut)

1.9104461669921875
```

On voit que le second s'est exécuté plus vite.

N.B : On utilise également les listes comprehension pour créer des *nested* listes, c'est-à-dire des listes dans d'autres listes.

## Exemple :

```
In [65]: ➜ listeTest=[[i for i in range(5)] for j in range(4)] #On genere une liste imbriquée de 4 listes avec chacune 5 éléments
        listeTest

Out[65]: [[0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4]]
```

# Dictionnaires Compréhension

```
In [65]: ┌─ listeTest=[[i for i in range(5)] for j in range(4)] #On genere une Liste imbriquee de 4 listes avec chacune 5 elements
         listeTest
```

```
Out[65]: [[0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4]]
```

```
In [66]: ┌─ prenoms=['Héritier', 'Jean', 'Didier']
         dict={k:v for k,v in enumerate(prenoms)} #On cree un dictionnaire comprehension
         print(dict)
```

```
{0: 'Héritier', 1: 'Jean', 2: 'Didier'}
```

```
In [67]: ┌─ prenoms=['Héritier', 'Jean', 'Didier']
         ages=[50,25,30]
         dico ={prenom:age for prenom, age in zip(prenoms, ages)} #On cree un dictionnaire comprehension
         print(dico)
```

```
{'Héritier': 50, 'Jean': 25, 'Didier': 30}
```

On peut aussi inclure des conditions dans des listes ou des dictionnaires comprehension :

```
In [69]: ┌─ #Exemple:
         prenoms=['Héritier', 'Jean', 'Didier']
         ages=[50,25,30]
         dico ={prenom:age for prenom, age in zip(prenoms, ages) if age > 45} #On cree un dictionnaire comprehension
         dico
```

```
Out[69]: {'Héritier': 50}
```

# La syntaxe du dictionnaire comprehension avec condition

dic={**prenom:age** for prenom, age in zip(prenoms, ages) **if age > 45**}



Ce qui doit etre inclus  
dans la boucle



La boucle



Condition

Fais ceci

[ prenom:age ]

Pour cette collection

for prenom, age in zip(prenoms,ages)

Dans cette situation

if age > 45]

# Tuple Comprehension

```
In [70]: ┆ tuple1=tuple((i**2 for i in range(15)))  
tuple1
```

```
Out[70]: (0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196)
```

# Exercice sur les fonctions/méthodes

## 1. Equation du second degré avec paramètres fixes

```

import math

def equation_second_degre(a, b, c):
    # Calcul du discriminant
    discriminant = b**2 - 4*a*c
    # Si le discriminant est négatif, il n'y a pas de solution réelle
    if discriminant < 0:
        #Le return None, None est utilisé pour indiquer qu'il n'y a pas de solution réelle
        #à L'équation du second degré lorsque le discriminant est négatif. Dans ce cas,
        #la fonction renvoie None pour les deux solutions x1 et x2,
        #ce qui signifie qu'il n'y a pas de solution réelle.
        return None, None
    # Calcul des solutions
    elif discriminant == 0:
        # Calcul de la solution unique
        x = -b / (2*a)
        return x, None
    else:
        x1 = (-b + math.sqrt(discriminant)) / (2*a)
        x2 = (-b - math.sqrt(discriminant)) / (2*a)
    return x1, x2
# Exemple d'utilisation
a = 1
b = -3
c = 1
x1, x2 = equation_second_degre(a, b, c)
if x1 is None and x2 is None:
    print("L'équation n'a pas de solution réelle.")
elif x2 is None:
    print("L'équation a une racine double x1=x2=", x1)
else:
    #Pour formater la chaîne de caractères avec les solutions x1 et x2,
    #vous devez utiliser la méthode format ou les f-strings(si vous utilisez Python 3.6 ou une version ultérieure).
    #Voici comment vous pourriez le faire avec format
    print("Les solutions de l'équation sont x1={} et x2={}".format(x1, x2))
    #avec f-strings ca serait
    #print(f"Les solutions de l'équation sont x1={x1} et x2={x2}")

```

Les solutions de l'équation sont x1=2.618033988749895 et x2=0.3819660112501051

# Exercice sur les fonctions/méthodes (Cont.)

## 2. Equation du second degré avec saisie de l'utilisateur des paramètres

```

import math

def equation_second_degre():
    a = float(input("Entrez le coefficient a: "))
    b = float(input("Entrez le coefficient b: "))
    c = float(input("Entrez le coefficient c: "))

    # Calcul du discriminant
    discriminant = b**2 - 4*a*c
    # Si le discriminant est négatif, il n'y a pas de solution réelle
    if discriminant < 0:
        return None, None
    # Calcul des solutions
    elif discriminant == 0:
        # Calcul de la solution unique
        x = -b / (2*a)
        return x, None
    else:
        x1 = (-b + math.sqrt(discriminant)) / (2*a)
        x2 = (-b - math.sqrt(discriminant)) / (2*a)
        return x1, x2
# Exemple d'utilisation
a = 1
b = -3
c = 1
x1, x2 = equation_second_degre()
if x1 is None and x2 is None:
    print("L'équation n'a pas de solution réelle.")
elif x2 is None:
    print("L'équation a une racine double x1=x2=", x1)
else:
    print(f"Les solutions de l'équation sont x1={x1} et x2={x2}")

```

Entrez le coefficient a: 2  
 Entrez le coefficient b: -3  
 Entrez le coefficient c: 1  
 Les solutions de l'équation sont x1=1.0 et x2=0.5