TPs

October 3, 2023

Contents

1 TP 1: Variables, Fonctions et Structure de control

1.1 Variables et Operations

En programmation, une variable est un espace de stockage nommé qui permet de stocker des données. En Python, un langage de programmation polyvalent et largement utilisé, il existe quatre grands types de variables qui nous permettent de stocker différents types de données. Comprendre ces types de variables est essentiel pour manipuler efficacement les données dans un programme Python. Les Quatre Types Principaux de Variables en Python

```
int (nombre entier) : Les variables de type "int" sont utilisées pour stocker des nombres entier

float (nombre décimal) : Les variables de type "float" sont utilisées pour stocker des nombres d

string (chaîne de caractères) : Les variables de type "string" sont utilisées pour stocker des s

bool (booléen) : Les variables de type "bool" sont utilisées pour stocker des valeurs booléennes
```

En Python, la déclaration d'une variable est simple. Vous pouvez attribuer une valeur à une variable en utilisant le signe égal (=). Par exemple, pour attribuer la valeur 42 à une variable nombre, vous feriez nombre = 42.

Il est important de noter que les noms de variables en Python doivent suivre certaines conventions, telles que ne pas commencer par un chiffre et ne contenir que des lettres, des chiffres et des underscores.

L'utilisation des variables est fondamentale en programmation, car elles nous permettent de stocker et de manipuler des données de manière dynamique. Dans ce cours, nous explorerons en détail chaque type de variable et comment les utiliser efficacement pour résoudre des problèmes de programmation.

Maintenant que nous avons une compréhension de base des variables en Python, explorons chacun de ces types en détail dans les sections suivantes.

[4]:
```
x = 3 # type int
y = 2.5 # type float
prenom = 'Pierre' # type string
z = True # type Bool

print(type(x),type(y), type(prenom),type(z))
```

```
<class 'int'> <class 'float'> <class 'str'> <class 'bool'>
```

1.1.1 Opérations Arithmétiques en Python

Les opérations arithmétiques sont un aspect fondamental de la programmation en Python. Elles permettent de réaliser des calculs mathématiques de base, ce qui est essentiel dans le développement de tout type d'application. Dans cette section, nous explorerons les opérations arithmétiques de base en Python, notamment l'addition, la soustraction, la multiplication, la division et la puissance.

Python offre une syntaxe simple et intuitive pour effectuer ces opérations, ce qui en fait un langage de programmation populaire pour les calculs mathématiques et scientifiques, ainsi que pour des applications plus générales.

Nous aborderons les sujets suivants dans cette section : - Addition (+) - Soustraction (-) - Multiplication (*) - Division (/) - Opérations avec des nombres entiers - Utilisation de parenthèses pour contrôler l'ordre des opérations - Utilisation de la fonction `pow()` pour calculer les puissances

Comprendre comment utiliser ces opérations est essentiel pour manipuler des données numériques et résoudre des problèmes mathématiques à l'aide de Python.

[3]:
```
# Opérations arithmétiques
print('x + y =', x + y)
print('x - y =', x - y)
print('x / y =', x / y)
print('x // y =', x // y) # division entiere (tres utile pour les tableaux Numpy)
print('x * y =', x * y)
print('x ^ y =', x ** y) # x puissance y
```

```
x + y = 5.5
x - y = 0.5
x / y = 1.2
x // y = 1.0
x * y = 7.5
x ^ y = 15.588457268119896
```

1.1.2 Opérations de Comparaison en Python

Les opérations de comparaison sont essentielles en programmation pour évaluer des conditions et prendre des décisions en fonction de ces évaluations. En Python, vous pouvez comparer des valeurs et des expressions pour déterminer si elles sont égales, plus grandes, plus petites ou d'autres relations. Dans cette section, nous explorerons les opérations de comparaison en Python.

Voici les principales opérations de comparaison que nous allons aborder : - Égalité (==) : Vérifie si deux valeurs sont égales. - Inégalité (!=) : Vérifie si deux valeurs ne sont pas égales. - Supériorité (>) : Vérifie si une valeur est strictement supérieure à une autre. - Infériorité (<) : Vérifie si une valeur est strictement inférieure à une autre. - Supériorité ou égalité (>=) : Vérifie si une valeur est supérieure ou égale à une autre. - Infériorité ou égalité (<=) : Vérifie si une valeur est inférieure ou égale à une autre.

Ces opérations de comparaison sont couramment utilisées pour créer des expressions conditionnelles, des boucles et d'autres structures de contrôle de flux dans vos programmes Python.

[4]:
```
# Opérations de comparaison
print('égalité :', x == y)
print('inégalité :', x != y)
print('inférieur ou égal :', x <= y)
print('supérieur ou égal :', x >= y)
```

```
égalité : False
inégalité : True
```

```
inférieur ou égal : False
supérieur ou égal : True
```

1.1.3 Opérations Logiques en Python

Les opérations logiques sont un élément clé de la programmation en Python, utilisées pour évaluer des conditions complexes et prendre des décisions basées sur ces conditions. Les opérations logiques permettent de combiner et de manipuler des valeurs booléennes (Vrai ou Faux) de manière à construire des expressions conditionnelles puissantes. Dans cette section, nous explorerons les opérations logiques en Python.

Les opérations logiques principales que nous allons aborder sont les suivantes : - ET logique (`and`) : Vérifie si deux conditions sont simultanément vraies. - OU logique (`or`) : Vérifie si au moins l'une des deux conditions est vraie. - NON logique (`not`) : Inverse une condition booléenne.

Ces opérations logiques sont utilisées pour créer des expressions conditionnelles complexes qui permettent de contrôler le flux de votre programme. Elles sont essentielles pour prendre des décisions, exécuter certaines parties du code en fonction de conditions, et créer des boucles conditionnelles.

[5]:
```python
# Opérations Logiques
print('ET :', False and True)
print('OU :', False or True)
print('OU exclusif :', False ^ True)
```

```
ET : False
OU : True
OU exclusif : True
```

Note : Les opérations de comparaison et de logique utilisées ensemble permettent de construire des structures algorithmiques de bases (if/esle, while, . . .)

[6]:
```python
# Table Logiques de AND
print('  A     B   ||  A and B   ')
print('-------------------------')
print('False True  ||  ', False and True)
print('False False ||  ', False and False)
print('True  True  ||  ', True and True)
print('True  False ||  ', False and False)
```

```
  A     B   ||  A and B
-------------------------
False True  ||   False
False False ||   False
True  True  ||   True
True  False ||   False
```

1.2 Fonctions

Les fonctions sont des éléments fondamentaux de la programmation en Python. Elles permettent d'organiser et de structurer le code en le divisant en blocs de code réutilisables. Une fonction est

un ensemble d'instructions qui effectuent une tâche spécifique et peuvent prendre des arguments en entrée pour personnaliser leur comportement. Les fonctions sont essentielles pour rendre le code plus lisible, modulaire et facile à entretenir.

Dans Python, une fonction est définie à l'aide du mot-clé def, suivi du nom de la fonction et de ses paramètres entre parenthèses. Elle peut avoir un corps de fonction qui contient les instructions à exécuter lorsque la fonction est appelée. Les fonctions peuvent également retourner des valeurs à l'aide du mot-clé return.

L'utilisation de fonctions permet de réduire la redondance du code en regroupant des actions similaires sous une seule fonction. Elles offrent également la possibilité de diviser un programme en morceaux plus petits et plus gérables, ce qui simplifie le débogage et la maintenance du code.

1.2.1 Les Fonctions Lambda en Python

Les fonctions lambda, également connues sous le nom de fonctions anonymes, sont un concept intéressant en Python. Contrairement aux fonctions définies avec def, les fonctions lambda sont des fonctions courtes et temporaires qui peuvent être créées sans avoir à nommer explicitement la fonction. Elles sont définies à l'aide du mot-clé lambda, suivi des paramètres entre parenthèses et d'une expression unique qui est évaluée et renvoyée comme résultat.

Présentation des Fonctions Lambda :

Les fonctions lambda sont souvent utilisées pour des opérations simples et ponctuelles, telles que les transformations de données dans une liste, les filtres ou les opérations mathématiques simples. Elles sont particulièrement utiles lorsque vous avez besoin d'une petite fonction pour effectuer une tâche spécifique, sans la complexité d'une fonction définie avec def.

Définition des Fonctions Lambda :

Une fonction lambda est définie comme suit :

```
lambda arguments: expression
```

```
arguments : Les paramètres que la fonction lambda prend en entrée, séparés par des virgules.
expression : L'expression unique qui est évaluée et renvoyée comme résultat lorsque la fonction
```

Voici un exemple simple d'une fonction lambda qui additionne deux nombres :

[7]:
```
# Exemple d'une fonction f(x) = x^2
f = lambda x : x**2

print(f(3))
```

```
9
```

[8]:
```
# Exemple d'une fonction g(x, y) = x^2 - y^2
g = lambda x, y : x**2 - y**2

print(g(4, 2))
```

```
12
```

[9]:
```
print(g(f(2), 2))
```

```
12
```

1.2.2 Fonction en Python

Une fonction en Python est une portion de code qui effectue une tâche spécifique ou une série de tâches. Les fonctions sont utilisées pour organiser et réutiliser du code en le divisant en blocs autonomes. Une fonction peut être conçue pour prendre des arguments en entrée, effectuer des opérations sur ces arguments, et renvoyer un résultat en sortie. Les fonctions sont définies à l'aide du mot-clé def, suivi du nom de la fonction et de ses paramètres entre parenthèses, le cas échéant.

Définition d'une Fonction :

Voici la structure de base de la définition d'une fonction en Python :

```
def nom_de_la_fonction(parametre1, parametre2, ...):
    # Instructions de la fonction
    # Ces instructions décrivent ce que la fonction fait
    # Elles peuvent inclure des calculs, des opérations, des boucles, etc.
    return resultat
```

```
nom_de_la_fonction : Le nom que vous choisissez pour votre fonction.
parametre1, parametre2, ... : Les paramètres (également appelés arguments) que la fonction peut
return resultat : Facultatif. Si vous souhaitez que votre fonction renvoie un résultat, utilisez
```

[10]:
```
# une fonction a un nom, prend des entrées (arguments) et les transforme pour␣
↪retourner un résultat

def nom_de_la_fonction(argument_1, argument_2):
   restultat = argument_1 + argument_2
   return restultat

nom_de_la_fonction(3, 2)
```

[10]:
```
5
```

[11]:
```
# Exemple concret : fonction qui calcul l'energie potentielle d'un corps

def e_potentielle(masse, hauteur, g=9.81):
  energie = masse * hauteur * g
  return energie

# ici g a une valeur par défaut donc nous ne sommes pas obligé de lui donner une␣
↪valeur
e_potentielle(masse=10, hauteur=10)
```

[11]:
```
981.0
```

a notion de docstring, ou chaîne de documentation, est un concept important en programmation Python. Il s'agit d'une convention utilisée pour documenter des fonctions, des classes, des modules ou des méthodes en fournissant une description claire et concise de ce qu'ils font, de leurs paramètres, de leurs valeurs de retour, et de toute autre information pertinente. Les docstrings sont des commentaires spéciaux dans le code Python qui sont accessibles via l'attribut **doc** d'un objet.

Voici quelques points clés pour comprendre les docstrings en Python :

```
Format des Docstrings : Les docstrings sont généralement encadrées par trois guillemets simples
```

[5]:
```python
def ma_fonction(parametre):
    """
    Description de la fonction.

    Args:
        parametre (type): Description du paramètre.

    Returns:
        type_de_retour: Description de la valeur de retour.
    """
    # Le code de la fonction
```

Contenu des Docstrings : Les docstrings doivent contenir des informations claires et utiles sur l'objet qu'ils documentent. Cela peut inclure :

```
Une brève description de ce que fait la fonction.
Une liste des paramètres, avec leur type et une description.
Une description de la valeur de retour de la fonction.
Des exemples d'utilisation.
```

Accès aux Docstrings : Vous pouvez accéder au contenu d'un docstring à l'aide de l'attribut **doc** de l'objet. Par exemple :

[6]:
```python
print(ma_fonction.__doc__)
```

```
    Description de la fonction.

    Args:
        parametre (type): Description du paramètre.

    Returns:
        type_de_retour: Description de la valeur de retour.
```

Cela affichera la documentation de la fonction ma_fonction à la console.

Utilité des Docstrings : Les docstrings sont utiles pour plusieurs raisons :

```
Ils aident les autres développeurs (ou vous-même) à comprendre rapidement comment utiliser une f
Ils servent de référence lors de la documentation du code.
Ils sont utilisés par des outils de documentation automatique tels que Sphinx pour générer des d
```

1.3 Structures de Contrôle en Programmation

Les structures de contrôle jouent un rôle central dans la programmation en Python (et dans de nombreux autres langages). Elles sont essentielles pour définir le flux d'exécution d'un programme, permettant ainsi de prendre des décisions, de répéter des tâches et d'organiser le code de manière logique. Dans cette section, nous explorerons trois principales structures de contrôle qui sont fondamentales pour la création d'algorithmes efficaces :

- **Les Blocks d'Instructions :** Ces structures délimitent des sections de code qui s'exécutent ensemble, souvent à l'intérieur d'une fonction ou d'une boucle. Les blocs d'instructions contribuent à organiser et à structurer votre code de manière compréhensible.
- **Les Instructions Conditionnelles (If / Else) :** Les instructions conditionnelles permettent de prendre des décisions dans un programme. Selon une condition donnée, le programme peut exécuter différentes parties de code. Le constructeur "If" permet d'exécuter une action si une condition est vraie, tandis que "Else" permet de définir une action alternative si la condition n'est pas remplie.
- **Les Instructions Itératives (For, While) :** Les instructions itératives sont utilisées pour répéter des actions plusieurs fois. La boucle "For" permet de parcourir un ensemble de données ou de répéter un bloc de code un nombre spécifique de fois. La boucle "While" répète un bloc de code tant qu'une condition reste vraie.

1.3.1 Les Blocks d'instructions

En programmation, un bloc d'instructions, aussi appelé "suite d'instructions" ou "bloc de code", est une séquence d'instructions qui sont regroupées ensemble et exécutées comme une seule unité. Les blocs d'instructions sont utilisés pour organiser et structurer le code de manière logique et cohérente. Ils jouent un rôle essentiel pour rendre un programme lisible et maintenable. Voici quelques points clés pour comprendre les blocs d'instructions :

```
Définition des Blocs d'Instructions : Un bloc d'instructions est généralement délimité par des s
```

[ ]:
```python
def ma_fonction():
    # Ceci est le début du bloc d'instructions
    instruction_1
    instruction_2
    # ...
    instruction_n
    # Ceci est la fin du bloc d'instructions
```

Les blocs d'instructions servent principalement à :

- Regrouper des instructions associées : Les instructions à l'intérieur d'un bloc ont souvent un objectif commun ou sont liées à une tâche spécifique.
- Définir la portée : Les variables déclarées à l'intérieur d'un bloc sont généralement visibles uniquement à l'intérieur de ce bloc, ce qui permet de limiter leur portée (scope).
- Faciliter la lisibilité : En regroupant des instructions similaires dans un bloc, le code devient plus lisible et plus facile à comprendre.

Usage Courant : Les blocs d'instructions sont couramment utilisés dans les fonctions et les méthodes, où ils regroupent le code spécifique à la fonction. Les boucles et les structures conditionnelles

utilisent également des blocs pour déterminer quelles parties du code doivent être répétées ou exécutées sous certaines conditions.

Emboîtement de Blocs : Il est important de noter que les blocs d'instructions peuvent être emboîtés, c'est-à-dire qu'un bloc peut contenir d'autres blocs. Cela permet de hiérarchiser et de structurer le code de manière complexe. Par exemple, une fonction peut contenir une boucle, et cette boucle peut elle-même contenir un bloc conditionnel, et ainsi de suite.

Indentation : En Python, contrairement à certains autres langages, l'indentation est cruciale pour définir les blocs d'instructions. Les lignes de code appartenant au même bloc doivent être indentées de la même manière, ce qui garantit la lisibilité du code

1.3.2 Les instructions conditionnelles :

Les instructions conditionnelles, également appelées structures de contrôle conditionnelles, sont un élément essentiel de la programmation. Elles permettent à un programme de prendre des décisions basées sur des conditions spécifiques. En Python, les instructions conditionnelles les plus couramment utilisées sont `if`, `elif` (else if), et `else`. Voici une explication détaillée des instructions conditionnelles :

1. **Instruction `if` :** L'instruction `if` permet d'exécuter un bloc de code uniquement si une condition spécifiée est évaluée comme vraie (True). La syntaxe de base est la suivante :

```
if condition:
    # Bloc de code à exécuter si la condition est vraie
```

Par exemple, vous pouvez utiliser une instruction if pour vérifier si un nombre est positif.

2. Instruction elif (Else If) : L'instruction elif est utilisée pour spécifier une condition alternative à vérifier si la première condition (if) n'est pas remplie. Elle permet de gérer plusieurs conditions différentes. Voici comment elle est utilisée :

```
if condition1:
    # Bloc de code à exécuter si condition1 est vraie
elif condition2:
    # Bloc de code à exécuter si condition2 est vraie
```

Par exemple, vous pouvez utiliser elif pour vérifier si un nombre est nul, positif ou négatif.

3. Instruction else : L'instruction else est utilisée pour spécifier un bloc de code à exécuter si aucune des conditions précédentes n'est vraie. Elle est souvent utilisée comme option de secours lorsque toutes les autres conditions échouent. Voici comment elle est utilisée :

```
if condition:
    # Bloc de code à exécuter si la condition est vraie
else:
    # Bloc de code à exécuter si la condition n'est pas vraie
```

Par exemple, dans l'exemple précédent, le bloc de code associé à else sera exécuté lorsque nombre est égal à 0.

[1]:
```
def test_du_signe(valeur):
  if valeur < 0:
    print('négatif')
  elif valeur == 0:
    print('nul')
  else:
    print('positif')
```

[4]:
```
(test_du_signe(-2))
(test_du_signe(2))
(test_du_signe(0))
```

```
négatif
positif
nul
```

Note importante : Une condition est respectée si et seulement si elle correspond au résultat **booléen True**.

[5]:
```
valeur = -2
print(valeur < 0) # le résultat de cette comparaison est True

if valeur < 0:
  print('négatif')
```

```
True
négatif
```

Cela permet de développer des algorithmes avec des mélanges d'opéations Logiques et d'opérations de comparaisons. Par exemple : *si il fait beau et qu'il faut chaud, alors j'irai me baigner*

[6]:
```
x = 3
y = -1
if (x>0) and (y>0):
  print('x et y sont positifs')
else:
  print('x et y ne sont pas tous les 2 positifs')
```

```
x et y ne sont pas tous les 2 positifs
```

1.3.3 2. Les instructions itératives For, while

Dans Python, les instructions itératives, notamment les boucles for et while, sont des structures de contrôle permettant de répéter des actions ou des séquences d'instructions un certain nombre de fois ou tant qu'une condition est remplie. Elles sont essentielles pour automatiser des tâches répétitives et pour parcourir des collections de données telles que des listes, des chaînes de caractères, ou des dictionnaires. Voici une présentation et une description détaillée de chacune de ces boucles : #### 1- Boucle for a boucle for est utilisée pour parcourir une séquence (comme une liste, une chaîne de caractères, un dictionnaire, etc.) et exécuter un bloc de code pour chaque élément de la séquence. La syntaxe de base de la boucle for est la suivante :

```
for element in sequence:
    # Bloc de code à exécuter pour chaque élément
```

- element : Une variable qui prend la valeur de chaque élément de la séquence à chaque itération.
- sequence : La séquence à parcourir, telle qu'une liste, une chaîne de caractères, ou une plage générée par range(). Exemple d'utilisation de la boucle for :

[13]:
```
# range(début, fin, pas) est une built-in fonction tres utile de python qui␣
↪retourne un itérable.
for i in range(0, 10):
  print(i)
```

```
0
1
2
3
4
5
6
7
8
9
```

[16]:
```
# on peut s'amuser a combiner cette boucle avec notre fonction de tout a l'heure.
for i in range(-10, 10, 2):
  print(i)
  test_du_signe(i)
```

```
-10
négatif
-8
négatif
-6
négatif
-4
négatif
-2
négatif
0
nul
2
positif
4
positif
6
positif
8
```

positif

1.3.4 2. Boucle While

La boucle while permet d'exécuter un bloc de code tant qu'une condition spécifiée reste vraie. Elle est utile lorsque vous ne savez pas à l'avance combien d'itérations seront nécessaires. La syntaxe de base de la boucle while est la suivante :

```
while condition:
    # Bloc de code à exécuter tant que la condition est vraie
```

condition : Une expression booléenne qui est évaluée à chaque itération. La boucle continue tant que cette condition est vraie.

[17]:
```
x = 0
while x < 10:
  print(x)
  x += 1 # incrémente x de 1 (équivalent de x = x+1)
```

```
0
1
2
3
4
5
6
7
8
9
```

1.4 Exercices

1.4.1 Exercices Partie 1

1. Modifiez la fonction e_potentielle définie plus haut pour retourner une valeur indiquant si l'energie calculée est supérieure ou inférieur a une **energie_limite** passée en tant que 4^{eme} argument
2. Créez une fonction Python qui prend deux nombres et un opérateur (+, -, *, /) en entrée. Effectuez l'opération arithmétique correspondante et renvoyez le résultat. Testez la fonction avec diverses valeurs d'entrée et d'opérateurs.
3. Écrivez une fonction Python qui calcule l'aire d'un cercle en fonction de son rayon. Utilisez la formule `aire = π * r^2`, où π (pi) est une constante. Permettez à l'utilisateur de saisir le rayon et affichez l'aire.
4. Créez un programme Python capable de convertir entre les températures Celsius et Fahrenheit. Écrivez des fonctions pour la conversion de Celsius en Fahrenheit et vice versa. Demandez à l'utilisateur de saisir la température et affichez la température convertie.

5. Écrivez une fonction Python qui vérifie si un entier donné est pair ou impair. Utilisez des déclarations conditionnelles (if/else) pour déterminer et renvoyer le résultat. Testez la fonction avec divers nombres.

6. Développez une fonction Python qui vérifie si une chaîne donnée est un palindrome (se lit de la même manière de gauche à droite et de droite à gauche). Supprimez les espaces et convertissez la chaîne en minuscules pour une vérification insensible à la casse. Renvoyez True s'il s'agit d'un palindrome et False sinon. Testez la fonction avec différents mots et phrases.

1.4.2 Exercices Partie 2

1. Implémentez la **suite de Fibonacci** [0, 1, 1, 2, 3, 5, 8, 13, 21, ...] qui part de 2 nombres a=0 et b=1, et qui calcule le nombre suivant en additionnant les 2 nombres précédents.

Indices : - Pour cet exercice vous aurez besoin d'une bouce **While** - Vous pouvez imprimer cette suite jusqu'a atteindre un nombre **n** que vous aurez choisit - dans python il est possible de mettre a jour 2 variables simultannément sur la meme lignes : **a, b = b, a+b**

2. Implémentez l'algorithme du jeu "trouver le nombre" qui est le suivant : Un nombre aléatoire sera généré et le but est de le trouver.

- À chaque itération, l'utilisateur essaie de deviner le nombre.
- si l'utilisateur devine le numéro correctement, le jeu se terminera et un message de félicitations apparaîtra
- si l'utilisateur donne un nombre supérieur, un message "inférieur" apparaîtra
- si l'utilisateur donne un nombre bas, un message "supérieur" apparaîtra

[13]:
```
#vous pouvez utiliser la function random
import random
x= random.randint(0,1000)
print(x)
```

```
901
```

3. Write a Python program that prints all the multiples of 5 between 1 and 50 (inclusive).

4. Create a Python function that calculates the factorial of a given positive integer. Ask the user for input and display the result.

5. Create a Python program that generates prime numbers within a specified range. Ask the user to input the range and display the prime numbers found.

2 TP 2: Structure de données

2.1 Listes et Tuples

En Python, les listes et les tuples sont deux structures de données fondamentales qui permettent de stocker et de gérer des collections d'éléments. Bien qu'ils partagent certaines similitudes, ils présentent des différences essentielles en termes de mutabilité, d'utilisation et de syntaxe. Voici une introduction aux listes et aux tuples en Python :

- Listes en Python :

Une liste est une collection ordonnée et mutable d'éléments. Les éléments d'une liste sont séparés par des virgules et sont entourés de crochets []. Les listes peuvent contenir des éléments de différents types (nombres, chaînes de caractères, autres listes, objets, etc.). Les listes peuvent être modifiées après leur création, ce qui signifie que vous pouvez ajouter, supprimer ou modifier des éléments. Pour accéder à un élément d'une liste, utilisez son index (l'indice), en commençant par 0 pour le premier élément. Exemple : ma_liste = [1, 2, 3, 'quatre', 'cinq']

- Tuples en Python :

  Un tuple est une collection ordonnée et immuable d'éléments. Les éléments d'un tuple sont séparés par des virgules et sont entourés de parenthèses (). Les tuples peuvent également contenir des éléments de différents types. Contrairement aux listes, les tuples ne peuvent pas être modifiés après leur création. Ils sont immuables. Pour accéder à un élément d'un tuple, utilisez son index de la même manière qu'avec les listes. Exemple : mon_tuple = (1, 2, 3, 'quatre', 'cinq')

- Utilisation Courante :

  Les listes sont couramment utilisées pour stocker des collections d'éléments lorsque vous avez besoin de modifier ces éléments au fil du temps. Par exemple, une liste peut représenter une liste de tâches à faire, des éléments dans un panier d'achat, etc. Les tuples sont souvent utilisés lorsque vous voulez vous assurer que les données restent constantes et ne doivent pas être modifiées accidentellement. Par exemple, les coordonnées géographiques d'un lieu, les dimensions d'un objet, etc.-

2.1.1 Création de Listes et de Tuples

une liste ou un tuple peuvent contenir tout types de valeurs (int, float, bool, string). On dit que ce sont des structures hétérogenes.

La différence entre les 2 est qu'une liste est **mutable** alors qu'un Tuple ne l'est pas (on ne peut pas le changer apres qu'il soit crée)

[1]:
```
# Listes
liste_1 = [1, 4, 2, 7, 35, 84]
villes = ['Paris', 'Berlin', 'Londres', 'Bruxelles']
nested_list = [liste_1, villes] # une liste peut meme contenir des listes ! On␣
 ↪appelle cela une nested list

#Tuples
tuple_1 = (1, 2, 6, 2)
```

[2]:
```
print(villes)
```

```
['Paris', 'Berlin', 'Londres', 'Bruxelles']
```

2.1.2 Indexing et Slicing

Indexing :

En Python, l'indexing fait référence à l'accès à un élément spécifique dans une séquence (comme une liste, une chaîne de caractères ou un tuple) en utilisant un numéro d'index. L'index est un entier

qui identifie la position de l'élément dans la séquence. Voici quelques points importants concernant l'indexing :

- L'indexing commence généralement à 0 pour le premier élément de la séquence. Par exemple, 0 est l'index du premier élément, 1 est l'index du deuxième élément, et ainsi de suite.
- Vous pouvez également utiliser des indices négatifs pour compter à partir de la fin de la séquence. Par exemple, - 1 représente le dernier élément, - 2 le deuxième élément en partant de la fin, et ainsi de suite.
- Pour accéder à un élément spécifique, utilisez la notation [indice] après le nom de la séquence. Par exemple, ma_liste[0] accède au premier élément d'une liste.
- Si l'indice spécifié est en dehors de la plage valide, une erreur "IndexError" sera levée.

Slicing :

Le slicing en Python consiste à extraire une partie d'une séquence en spécifiant une plage d'indices. Il vous permet de créer une nouvelle séquence à partir d'une partie de la séquence d'origine. Voici comment fonctionne le slicing :

- Utilisez la notation [début:fin] après le nom de la séquence pour spécifier la plage d'indices que vous souhaitez extraire.
- Le slicing inclut l'élément au niveau de l'index de début, mais exclut l'élément au niveau de l'index de fin. Cela signifie que ma_liste[1:4] extrait les éléments à l'indice 1, 2 et 3, mais pas l'élément à l'indice 4.
- Si vous omettez l'indice de début, le slicing commencera à partir du premier élément. Si vous omettez l'indice de fin, le slicing ira jusqu'au dernier élément inclus.
- Vous pouvez également spécifier un pas en ajoutant un troisième paramètre [début:fin:pas]. Par exemple, ma_liste[0:5:2] extraira les éléments aux indices 0, 2 et 4.
- Le slicing fonctionne de la même manière pour les chaînes de caractères et les tuples.

[7]:
```
# INDEXING

print('séquence complete:', villes)
print('index 0:', villes[0])
print('index 1:', villes[1])
print('dernier index (-1):', villes[-1])
```

```
séquence complete: ['Paris', 'Berlin', 'Londres', 'Bruxelles']
index 0: Paris
index 1: Berlin
dernier index (-1): Bruxelles
```

[9]:
```
# SLICING [début (inclus) : fin (exclus) : pas]

print('séquence complete:', villes)
print('index 0-2:', villes[0:3])
print('index 1-2:', villes[1:3])
print('ordre inverse:', villes[::-1])
```

```
séquence complete: ['Paris', 'Berlin', 'Londres', 'Bruxelles']
index 0-2: ['Paris', 'Berlin', 'Londres']
```

```
index 1-2: ['Berlin', 'Londres']
ordre inverse: ['Bruxelles', 'Londres', 'Berlin', 'Paris']
```

2.1.3 Actions utiles sur les listes

- append(element): append is a list method that adds a specified element to the end of the list. It is commonly used when you want to add a single element to the existing list.
- insert(index, element): insert is a list method that allows you to insert an element at a specific index (position) within the list. You provide both the index where you want to insert the element and the element itself.
- extend(iterable): extend is a list method used to add multiple elements to the end of an existing list. It takes an iterable (such as a list, tuple, or string) as an argument and appends all its elements to the end of the list.
- sort(reverse=False): sort is a list method that arranges the elements of the list in a specific order. By default, it sorts the list in ascending order (from lowest to highest). You can also use reverse=True to sort in descending order.
- count(element): count is a list method that counts the number of occurrences of a specific element in the list. It is useful when you want to determine how many times a particular value appears in the list.

[3]:
```python
villes = ['Paris', 'Berlin', 'Londres', 'Bruxelles'] # liste initiale
print(villes)

villes.append('Dublin') # Rajoute un élément a la fin de la liste
print(villes)

villes.insert(2, 'Madrid') # Rajoute un élément a l'index indiqué
print(villes)

villes.extend(['Amsterdam', 'Rome']) # Rajoute une liste a la fin de notre liste
print(villes)

print('longeur de la liste:', len(villes)) #affiche la longueur de la liste

villes.sort(reverse=False) # trie la liste par ordre alphabétique / numérique
print(villes)

print(villes.count('Paris')) # compte le nombre de fois qu'un élément apparait␣
↪dans la liste
```

```
['Paris', 'Berlin', 'Londres', 'Bruxelles']
['Paris', 'Berlin', 'Londres', 'Bruxelles', 'Dublin']
['Paris', 'Berlin', 'Madrid', 'Londres', 'Bruxelles', 'Dublin']
['Paris', 'Berlin', 'Madrid', 'Londres', 'Bruxelles', 'Dublin', 'Amsterdam',
'Rome']
longeur de la liste: 8
```

```
['Amsterdam', 'Berlin', 'Bruxelles', 'Dublin', 'Londres', 'Madrid', 'Paris',
'Rome']
1
```

Les listes et les tuples fonctionnent en harmonies avec les structures de controle **if/else** et **For**

[23]:
```
if 'Paris' in villes:
  print('oui')
else:
  print('non')
```

```
oui
```

[24]:
```
for element in villes:
  print(element)
```

```
Amsterdam
Berlin
Bruxelles
Dublin
Londres
Madrid
Paris
Rome
```

2.1.4 enumerate en Python :

La fonction enumerate en Python est une fonction intégrée qui est utilisée pour énumérer les éléments d'une séquence, comme une liste ou une chaîne de caractères, tout en fournissant leur indice (position) dans la séquence. Elle est couramment utilisée dans les boucles pour accéder à la fois à l'élément et à son indice. Voici une explication en français :

```
enumerate est une fonction qui prend en entrée une séquence (comme une liste ou une chaîne de ca
L'indice commence généralement à 0 pour le premier élément de la séquence, puis s'incrémente à c
La fonction enumerate est utile lorsque vous avez besoin à la fois de la valeur de l'élément et
Elle permet d'améliorer la lisibilité du code en évitant d'utiliser une variable de compteur pou
```

[4]:
```
for index, element in enumerate(villes):
  print(index, element)
```

```
0 Amsterdam
1 Berlin
2 Bruxelles
3 Dublin
4 Londres
5 Madrid
6 Paris
7 Rome
```

2.1.5 zip en Python :

La fonction zip en Python est une fonction intégrée qui combine plusieurs séquences (telles que des listes ou des tuples) en une seule séquence, en associant les éléments correspondants par position. Voici une explication en français :

```
zip prend en entrée une ou plusieurs séquences de même longueur et renvoie un objet itérable qui
Les séquences doivent avoir la même longueur, sinon zip s'arrêtera lorsque la séquence la plus c
La fonction zip est couramment utilisée pour regrouper des données connexes à partir de différen
Elle permet de simplifier le code en évitant l'utilisation de boucles pour parcourir simultanéme
```

[7]:
```
liste_2 = [312, 52, 654, 23, 65, 12, 678]
liste_3 = [1, 2, 3, 4, 5, 6, 7]
for element_1, element_2,element_3 in zip(villes, liste_2,liste_3):
  print(element_3," - ",element_1, element_2)
```

```
1  -  Amsterdam 312
2  -  Berlin 52
3  -  Bruxelles 654
4  -  Dublin 23
5  -  Londres 65
6  -  Madrid 12
7  -  Paris 678
```

2.2 Dictionnaires

Les dictionnaires sont des structures de controle **non-ordonnées**, c'est-a-dire que les valeurs qu'ils contiennent ne sont pas rangée selon un index, mais suivant une **clef unique**.

Une utilisation parfaite des dictionnaires est pour regrouper ensemble des "variables" dans un meme conténaire. (ces variables ne sont pas de vraies variables, mais des **keys**).

On peut par exemple crée un dictionnaire inventaire qui regroupe plusieurs produits (les clefs) et leur quantités (les valeurs)

1. Dict.values() :

   Dict.values() est une méthode de dictionnaire qui renvoie une vue d'ensemble (view) des valeurs du dictionnaire. Cette vue d'ensemble contient toutes les valeurs du dictionnaire, sans les clés associées. Il est couramment utilisé pour accéder à toutes les valeurs d'un dictionnaire sans avoir besoin de parcourir les clés.

2. Dict.keys() :

   Dict.keys() est une méthode de dictionnaire qui renvoie une vue d'ensemble des clés du dictionnaire. Cette vue d'ensemble contient toutes les clés du dictionnaire, sans les valeurs associées. Elle est utilisée pour accéder à toutes les clés du dictionnaire, ce qui est utile lorsque vous avez besoin de parcourir les clés ou de vérifier si une clé spécifique existe dans le dictionnaire.

3. Dict.fromkeys() :

   Dict.fromkeys(iterable, valeur_par_défaut) est une méthode de classe qui crée un nouveau dictionnaire avec des clés à partir des éléments de l'itérable fourni. Chaque clé est associée à

la même valeur par défaut spécifiée. Si aucune valeur par défaut n'est fournie, les clés sont associées à None. Elle est utilisée pour initialiser un dictionnaire avec des clés à partir d'une séquence donnée, par exemple une liste ou un tuple.

4. Obtenir des informations (inventaire['clé'] = 30 ou inventaire.get('pommes')) :

   Vous pouvez obtenir des informations à partir d'un dictionnaire en utilisant la notation de crochets (dictionnaire['clé']) pour accéder à la valeur associée à une clé spécifique. L'opération dictionnaire.get('clé') permet également d'obtenir la valeur associée à une clé, mais elle est plus sûre car elle ne génère pas d'erreur si la clé n'existe pas dans le dictionnaire. Elle renvoie plutôt None ou une valeur par défaut spécifiée.

5. Dict.pop("clé") :

   Dict.pop("clé") est une méthode qui supprime l'élément associé à la clé spécifiée dans le dictionnaire et renvoie la valeur de cet élément. Si la clé n'existe pas dans le dictionnaire, elle peut renvoyer une erreur si une valeur par défaut n'est pas spécifiée.

6. Dict.keys(), Dict.values(), Dict.items() :

   Dict.keys() renvoie une vue d'ensemble des clés du dictionnaire. Dict.values() renvoie une vue d'ensemble des valeurs du dictionnaire. Dict.items() renvoie une vue d'ensemble des paires clé-valeur du dictionnaire. Ces méthodes sont couramment utilisées pour parcourir ou examiner les clés, les valeurs ou les paires clé-valeur dans un dictionnaire.

[1]:
```
inventaire = {'pommes': 100,
              'bananes': 80,
              'poires': 120}
```

[2]:
```
inventaire.values()
```

[2]:
```
dict_values([100, 80, 120])
```

[3]:
```
inventaire.keys()
```

[3]:
```
dict_keys(['pommes', 'bananes', 'poires'])
```

[4]:
```
len(inventaire)
```

[4]:
```
3
```

Voici comment ajouter une association key/value dans notre dictionnaire (attention si la clef existe déja elle est remplacée)

[2]:
```
inventaire['abricots'] = 30
print(inventaire)
```

```
{'pommes': 100, 'bananes': 80, 'poires': 120, 'abricots': 30}
```

Attention : si vous cherchez une clef qui n'existe pas dans un dictionnaire, python vous retourne une erreur. Pour éviter cela, vous pouvez utiliser la méthode **get()**

[6]:
```
inventaire.get('peches') # n'existe pas
```

[7]:
```
inventaire.get('pommes') # pomme existe
```

[7]:
```
100
```

[22]:
```
Animal_list =('dog','cat','bird','fish')
Animal_number = 0
MyAnimals={

}
MyAnimals.fromkeys(Animal_list,Animal_number)
```

[22]:
```
{'dog': 0, 'cat': 0, 'bird': 0, 'fish': 0}
```

la méthode **pop()** permet de retirer une clef d'un dictionnaire tout en retournant la valeur associée a la clef.

[8]:
```
abricots = inventaire.pop("abricots")
print(inventaire) # ne contient plus de clef abricots
print(abricots) # abricots contient la valeur du dictionnaire
```

```
{'pommes': 100, 'bananes': 80, 'poires': 120}
30
```

Pour utiliser une boucle for avec un dictionnaire, il est utile d'utiliser la méthode **items()** qui retourne a la fois les clefs et les valeurs

[5]:
```
for key in inventaire.keys():
  print(f'this is the key: {key}')
print('--------------')
for value in inventaire.values():
  print('this is the values: '.format(value))
print('--------------')
for key, value in inventaire.items():
  print(f'this is key-value: {key},{value}' )
```

```
this is the key: pommes
this is the key: bananes
this is the key: poires
this is the key: abricots
--------------
this is the values:
this is the values:
this is the values:
this is the values:
--------------
this is key-value: pommes,100
this is key-value: bananes,80
```

```
this is key-value: poires,120
this is key-value: abricots,30
```

2.3 Les ensembles (Sets)

En Python, un ensemble (ou "set" en anglais) est une collection non ordonnée d'éléments uniques. Contrairement à d'autres structures de données comme les listes et les tuples, les ensembles ne permettent pas de stocker des éléments en doublon. Voici une explication détaillée sur les ensembles en Python :

1. Collection Non Ordonnée : Les ensembles ne maintiennent pas l'ordre d'insertion des éléments. Par conséquent, il n'y a pas d'indice associé à chaque élément dans un ensemble. L'ordre des éléments dans un ensemble peut varier lorsque vous parcourez l'ensemble.
2. Éléments Uniques : Les ensembles ne permettent pas de stocker plusieurs fois le même élément. Si vous tentez d'ajouter un élément déjà présent dans l'ensemble, il ne sera pas ajouté une seconde fois.
3. Définition d'un Ensemble : Pour créer un ensemble en Python, vous pouvez utiliser des accolades {} ou la fonction set(). Par exemple : `python mon_set = {1, 2, 3}` ou `mon_set = set([1, 2, 3])`.
4. Opérations sur les Ensembles : Les ensembles prennent en charge un ensemble d'opérations courantes, notamment l'ajout d'éléments, la suppression d'éléments, la vérification de l'appartenance d'un élément, l'union de deux ensembles, l'intersection, la différence, etc.
5. Utilisations Courantes : Les ensembles sont couramment utilisés pour effectuer des opérations ensemblistes, telles que la recherche d'éléments uniques dans une liste, la déduplication de données, la vérification de l'appartenance d'un élément à un ensemble de données, etc.
6. Immuabilité des Éléments : Les éléments stockés dans un ensemble doivent être immuables, c'est-à-dire qu'ils ne peuvent pas être modifiés une fois qu'ils sont ajoutés à l'ensemble. Les types de données immuables courants comprennent les nombres, les chaînes de caractères et les tuples.
7. Itérabilité : Vous pouvez parcourir les éléments d'un ensemble à l'aide d'une boucle for, mais rappelez-vous que l'ordre des éléments n'est pas garanti.
8. Méthodes utiles : Les ensembles disposent de nombreuses méthodes utiles, telles que add() pour ajouter un élément, remove() pour supprimer un élément, union() pour l'union de deux ensembles, intersection() pour l'intersection, difference() pour la différence, etc.

2.4 Exercice

1. Transformer le code suivant qui donne la **suite de Fibonacci** pour enregistrer les résultats dans une liste et retourner cette liste a la fin de la fonction

```python
# Exercice :
def fibonacci(n):
    a = 0
    b = 1
    while b < n:
```

```
    a, b = b, a+b
    print(a)
```

2. Implémentez une fonction *trier(classeur, valeur)* qui place une valeur dans un dictionnaire en fonction de son signe

```
classeur = {'négatifs':[],
            'positifs':[]
            }
```

3. Implémentez un code qui permet de lire les valeur du classeur parameters

```
parameters = {'Ws':[1,2,3,4,8],
            'Bs':[1,2,3,5,6]
            }
```

4. Écrivez un programme Python qui permet à l'utilisateur de saisir une liste de nombres, puis ajoute un nombre supplémentaire à la fin de la liste. Ensuite, le programme doit compter combien de fois ce nombre supplémentaire apparaît dans la liste et afficher le résultat.

5. Créez un tuple contenant une série de nombres. Écrivez un programme Python qui génère un nouveau tuple contenant la somme cumulative des nombres du tuple d'origine. Par exemple, si le tuple initial est (1, 2, 3, 4), le nouveau tuple devrait être (1, 3, 6, 10).

6. Créez un dictionnaire de contacts en utilisant les noms comme clés et les numéros de téléphone comme valeurs. Écrivez un programme Python qui permet à l'utilisateur d'ajouter de nouveaux contacts, de rechercher un numéro de téléphone par nom, de supprimer un contact existant et d'afficher la liste complète des contacts.

7. Écrivez un programme Python qui prend une liste de nombres et supprime tous les éléments de la liste qui sont plus petits qu'une valeur seuil spécifiée par l'utilisateur. Affichez la liste résultante après la suppression.

8. Créez deux tuples contenant des noms d'étudiants. Écrivez un programme Python qui fusionne ces deux tuples en un seul, puis trie le tuple résultant par ordre alphabétique des noms. Affichez le tuple trié.

3 TP 3: List Comprehension et fonctions intégrées

3.1 List Comprehension

La compréhension de liste, souvent appelée "list comprehension" en anglais, est une manière concise et puissante de créer des listes en Python. Elle permet de générer rapidement une nouvelle liste en appliquant une expression à chaque élément d'une séquence (comme une liste, un tuple, ou une chaîne de caractères) ou en utilisant des conditions pour filtrer les éléments de la séquence d'origine. Voici une explication détaillée de la compréhension de liste :

Syntaxe de base de la compréhension de liste :

La syntaxe générale de la compréhension de liste est la suivante :

```
nouvelle_liste = [expression for élément in séquence if condition]
```

```
nouvelle_liste : C'est la nouvelle liste que vous allez créer.
```

```
expression : C'est une expression Python qui sera évaluée pour chaque élément de la séquence.
élément : C'est une variable temporaire qui prend la valeur de chaque élément de la séquence à c
séquence : C'est la séquence d'origine (liste, tuple, chaîne de caractères, etc.) que vous parco
condition (optionnelle) : C'est une condition qui filtre les éléments de la séquence. Seuls les
```

Exemple d'utilisation :

Voici un exemple simple de compréhension de liste pour créer une liste des carrés des nombres de 0 à 9 :

```
carres = [x**2 for x in range(10)]
```

Dans cet exemple, x prend la valeur de chaque nombre de 0 à 9, l'expression x**2 calcule le carré de chaque nombre, et la nouvelle liste carres contient les carrés des nombres.

Utilisation de la condition :

Vous pouvez également utiliser une condition pour filtrer les éléments de la séquence d'origine. Par exemple, pour créer une liste des nombres pairs entre 0 et 9 :

```
nombres_pairs = [x for x in range(10) if x % 2 == 0]
```

Dans cet exemple, seuls les nombres pairs (ceux pour lesquels x % 2 == 0 est vrai) sont inclus dans la nouvelle liste nombres_pairs.

Avantages de la compréhension de liste :

```
Concision : La compréhension de liste permet d'exprimer des opérations de création de liste de m

Performance : Elle est généralement plus rapide que l'utilisation de boucles for traditionnelles

Clarté : Elle améliore la lisibilité du code en évitant d'avoir à déclarer une liste vide et à a
```

Les deux code ci-dessous effectuent chacun la meme opération. On peut voir (grace a la commande %%time) que le temps d'execution avec liste comprehenion est bien inférieur au temps d'execution avec la méthode append()

[5]:
```
%%time
liste = []
for i in range(1000000):
  liste.append(i**2)
```

```
CPU times: user 406 ms, sys: 22.3 ms, total: 428 ms
Wall time: 526 ms
```

[6]:
```
%%time
liste = [i**2 for i in range(1000000)]
```

```
CPU times: user 334 ms, sys: 27.7 ms, total: 361 ms
Wall time: 427 ms
```

[7]:
```
liste = [[i**2 for i in range(0,10,2)] for i in range(3)]
print(liste)
```

```
[[0, 4, 16, 36, 64], [0, 4, 16, 36, 64], [0, 4, 16, 36, 64]]
```

On peut rajouter des conditions **if** dans les listes comprehension, par exemple :

[8]:
```
liste = [i**2 for i in range(100000) if (i % 2) == 0] # calcule i**2 seulement␣
↪pour les nombres pairs.

print(liste[:10]) #affiche les 10 premiers éléments de la liste
```

```
[0, 4, 16, 36, 64, 100, 144, 196, 256, 324]
```

[9]:
```
names=['adel','omar','moktar','ali']
dico1={name:idx for idx,name in enumerate(names)}
print((dico1))
print(dico1.keys())
print(dico1.values())
```

```
{'adel': 0, 'omar': 1, 'moktar': 2, 'ali': 3}
dict_keys(['adel', 'omar', 'moktar', 'ali'])
dict_values([0, 1, 2, 3])
```

[10]:
```
ages=[30,23,25,26]
dico2={names:ages for names,ages in zip(names,ages)}
print((dico2))
print(dico2.keys())
print(dico2.values())
```

```
{'adel': 30, 'omar': 23, 'moktar': 25, 'ali': 26}
dict_keys(['adel', 'omar', 'moktar', 'ali'])
dict_values([30, 23, 25, 26])
```

[16]:
```
ages=[30,23,25,26]
dico2={names:ages for names,ages in zip(names,ages) if ages< 30}
print((dico2))
print(dico2.keys())
print(dico2.values())
```

```
{'omar': 23, 'moktar': 25, 'ali': 26}
dict_keys(['omar', 'moktar', 'ali'])
dict_values([23, 25, 26])
```

3.2 Built-in Functions

Python contient un grand nombre de fonctions intégrées tres utiles a connaitre. Cela vous permet de construire des codes plus rapidement, sans avoir a développer vos propres fonctions pour les taches les plus basique. Dans ce notebook, je vous montre les plus importantes :

3.2.1 Fonction de bases

Utiles en toute circonstance !

[2]:
```
import numpy as np
x = np.pi
print(abs(x)) # valeur absolue
print(round(x,8)) # arrondi
```

```
3.141592653589793
3.14159265
```

[3]:
```
liste = [-2, 3, 1, 0, -4]

print(min(liste)) # minimum
print(max(liste)) # maximum
print(len(liste)) # longueur
print(sum(liste)) # somme des éléments
```

```
-4
3
5
-2
```

[4]:
```
liste = [False, False, True]

print(any(liste)) # y-a-t'il au moins un élément True ?
print(all(liste)) # est-ce-que tous les éléments sont True ?
```

```
True
False
```

3.2.2 Fonction de conversion

Il peut etre tres utile de convertir une variable d'un type a un autre (par exemple pour faire des calculs). Pour cela, on dispose des fonctions int(), str() et float().

La fonction **type()** est tres utile pour inspecter les types de nos variables

[5]:
```
age = '32'
type(age)
```

[5]:
```
str
```

[6]:
```
age = int(age)
type(age)
```

[6]:
```
int
```

[7]:
```
age + 10
```

[7]:
```
42
```

[8]:
```
float(x)
type(x)
```

[8]:
```
float
```

On peut également convertir des listes en tuples, ou des tableaux Numpy (que l'on verra par la suite) en liste...

[9]:
```
tuple_1 = (1, 2, 3, 4)

liste_1 = list(tuple_1) # convertir un tuple en liste
print(liste_1)
type(liste_1)
```

```
[1, 2, 3, 4]
```

[9]:
```
list
```

[10]:
```
print(bin(15),bin(32))
print(oct(15),oct(32))
print(hex(15),hex(32))
```

```
0b1111 0b100000
0o17 0o40
0xf 0x20
```

3.2.3 La fonction input()

La fonction input() en Python est une fonction intégrée qui permet à un programme d'obtenir des données en entrée depuis l'utilisateur via le clavier. Elle est principalement utilisée pour interagir avec l'utilisateur en demandant des informations ou des réponses à des questions. Voici une brève explication de la fonction input() :

- Lorsque la fonction input() est appelée, elle affiche un message d'invite (généralement une chaîne de caractères) à l'utilisateur, qui apparaît dans la console.
- L'utilisateur peut alors saisir des données depuis le clavier, suivi de la touche "Entrée".
- La fonction input() attend que l'utilisateur entre des données et retourne une chaîne de caractères (string) contenant les données saisies. L'application de la fonction input() est très utile pour créer des programmes interactifs, des questionnaires, des formulaires de saisie, etc.
- Il est important de noter que la fonction input() renvoie toujours une chaîne de caractères (string). Si vous avez besoin de manipuler des données numériques, vous devrez les convertir en types de données appropriés, tels que des entiers ou des nombres à virgule flottante, à l'aide des fonctions de conversion telles que int() ou float().

[12]:
```
age = input('quel age avez-vous ?')
```

```
quel age avez-vous ?30 ans
```

[14]:
```
type(age) # age est de type string. il faut penser a le convertir si on désire␣
↪faire un calcul avec
```

[14]:
```
str
```

3.2.4 La fonction format()

En Python, la méthode format() est utilisée pour formater des chaînes de caractères en y insérant des valeurs ou des variables dans des emplacements spécifiés, appelés “emplacements de remplacement” ou “placeholders”. Elle permet de créer des chaînes de caractères dynamiques en insérant des valeurs dans des positions prédéfinies. Voici une explication en français :

- La méthode format() est appelée sur une chaîne de caractères (la chaîne de format) et accepte un ou plusieurs arguments qui seront insérés dans la chaîne aux emplacements spécifiés.
- Les emplacements de remplacement sont indiqués par des accolades {} dans la chaîne de format. Par exemple, “Bonjour, {} !” est une chaîne de format avec un emplacement de remplacement.
- Vous pouvez spécifier l’ordre dans lequel les valeurs seront insérées en utilisant des indices numériques à l’intérieur des accolades. Par exemple, “{} {} {}” utilisera les trois premiers arguments dans cet ordre.
- Vous pouvez également nommer les emplacements de remplacement pour une utilisation plus explicite. Par exemple, “Bonjour, {prenom} !” utilise un emplacement nommé {prenom}.
- Les valeurs à insérer sont passées comme arguments à la méthode format(), soit dans l’ordre, soit en utilisant des noms de paramètres si des emplacements nommés sont utilisés.
- La méthode format() effectue automatiquement la conversion de types, ce qui signifie que vous pouvez insérer des valeurs de différents types (entiers, nombres à virgule flottante, chaînes de caractères, etc.) dans la chaîne de format.

[15]:
```
x = 25
ville = 'Paris'

message = 'il fait {} degrés a {}'.format(x, ville)
print(message)
```

```
il faut 25 degrés a Paris
```

[17]:
```
message = f'il fait {x} degrées a {ville}'
print(message)
```

```
il fait 25 degrées a Paris
```

oici quelques astuces et techniques que vous pouvez utiliser avec la méthode format() en Python pour formater vos chaînes de caractères de manière plus efficace et lisible :

- Alignement du texte : Vous pouvez spécifier l’alignement du texte à l’intérieur de l’emplacement de remplacement en utilisant les caractères $<$, $>$, ou $\hat{}$. Par exemple : “{:$>$10}”.format(“texte”) alignera le texte à droite sur une largeur de 10 caractères.

[10]:
```
fruits=['banana','grapes','strawberries','pear','peach']
for fruit in fruits:
    print("-{:>15}".format(fruit))
```

```
-         banana
-         grapes
-    strawberries
-           pear
-          peach
```

- Remplacement conditionnel : Vous pouvez conditionnellement choisir quelle valeur insérer en fonction d'une condition. Par exemple : "Bonjour, {} !".format(nom if condition else "Cher utilisateur").

[12]:
```
a=3
condition=True
nom='Omar'
print("Bonjour, {} !".format(nom if condition else "Cher utilisateur"))

condition=False

print("Bonjour, {} !".format(nom if condition else "Cher utilisateur"))
```

```
Bonjour, Omar !
Bonjour, Cher utilisateur !
```

Répétition d'un motif : Vous pouvez répéter un motif dans un emplacement de remplacement. Par exemple : "-" * 20 crée une chaîne composée de 20 tirets.

[15]:
```
"-"*20,"*"*20
```

[15]:
```
('--------------------', '**********')
```

3.2.5 La fonction open()

Cette fonction est l'une des plus utile de Python. Elle permet d'ouvrir n'importe quel fichier de votre ordinateur et de l'utiliser dans Python. Différents modes existent : - le mode 'r' : lire un fichier de votre ordinateur - le mode 'w' : écrire un fichier sur votre ordinateur - le mode 'a' : (append) ajouter du contenu dans un fichier existant

[5]:
```
f = open('./assets/text.txt', 'w') # ouverture d'un objet fichier f
f.write('hello')
f.close() # il faut fermer notre fichier une fois le travail terminé
```

[6]:
```
f = open('./assets/text.txt', 'r')
print(f.read())
f.close()
```

```
hello
```

Dans la pratique, on écrit plus souvent **with open() as f** pour ne pas avoir a fermer le fichier une fois le travail effectué :

[7]:
```
with open('./assets/text.txt', 'r') as f:
    print(f.read())
```

```
hello
```

3.3 Exercice

1. Le code ci-dessous permet de créer un fichier qui contient les nombres carrée de 0 jusqu'a 19. L'exercice est d'implémenter un code qui permet de lire ce fichier et d'écrire chaque ligne dans une liste.

Note_1 : la fonction **read().splitlines()** sera tres utile

Note_2 : Pour un meilleur résultat, essayer d'utiliser une liste comprehension !

[8]:
```
# Ce bout de code permet d'écrire le fichier
with open('./assets/fichier.txt', 'w') as f:
    for i in range(0, 20):
        f.write(f'{i}: {i**2} \n')
    f.close()

# Écrivez ici le code pour lire le fichier et enregistrer chaque lignes dans une␣
 ↪liste.
```