Définitions

Les fonctions sont des **instructions préalablement définies** et qui s'appliqueront dans le code lorsque la fonction sera **appelée**.

Tout comme pour les phases de création d'une variable, le cycle de vie d'une fonction se constitue de plusieurs étapes.

- **Définition de la fonction**: Le code de la fonction est définie, en précisant ses éventuels paramètres, le code que la fonction exécute, ainsi que son éventuelle valeur de retour. Ce code est complètement isolé de l'extérieur.
- **Appel de la fonction**: Le moment où le code préalablement défini par la fonction est exécuté et a un impact dans le fonctionnement du script.

```
# Définition de la fonction
def add(a, b):
    return a + b

# Appel de la fonction
result = add(2, 3)
print(result)
```

Le mot clé `return`

Le mot clé `return` permet de signaler à la fonction qu'elle doit **retourner une valeur**. Une fois le `return` exécuté, nous **sortons** de la fonction et toute ligne de code qui se situerait à la suite de cette instruction `return` ne sera **jamais exécutée**.

```
def increment(x):
    x += 1
    return x
    print("Je ne serai jamais éxécuté :(")

print(increment(2))
```

Valeur de retour d'une fonction

Une fonction retourne **toujours** quelque chose, même si il n'y a pas de `return` dans sa définition. Ainsi la valeur retournée dans ce cas sera `None`.
-> `None`, c'est la valeur du rien en Python.

```
def do_nothing():
    pass # le mot clé pass signifie que cette ligne de code ne donne aucune instruction

print(do_nothing())
```

Les paramètres d'une fonction

Une fonction peut avoir de 0 à plusieurs paramètres. Ces paramètres représentent des **abstractions** qui symbolisent les futurs objets concrets que la fonction va rencontrer lorsque celle-ci sera **appelée**.

```
def multiply(a, b): # a et b n'existent pas, ce sont des abstractions
    return a * b   # pour tout a et b, la fonction multiply me retournera a * b

result = multiply(2, 3) # 2 et 3 sont des objets concrets lors de l'appel de la fonction
print(result) # 6
```

Par ailleurs, on peut définir des **paramètres par défaut**, qui seront **optionnels**, et qui prendront une certaine valeur si ils ne sont pas présents lors de la phrase d'appel de la fonction.

```
def dire_bonjour(name = "mystérieux inconnu"):
    return f"Coucou {name} !"

print(dire_bonjour("Stéphane")) # Coucou Stéphane !
print(dire_bonjour())           # Coucou mystérieux inconnu !
```

⚠️ Si vous définissez des paramètres par défaut avec des paramètres obligatoires, vous devez vous assurer que le paramètres par défaut sont déclarés après tous les paramètres requis.

Hiérarchie du code

Ordre de déclaration et d'appel

L'interpréteur Python va exécuter le script de haut en bas. Ainsi la définition d'une fonction doit toujours se produire **avant** l'appel de cette fonction.

```
dire_coucou()


def dire_coucou():
    print("coucou !") # name 'dire_coucou' is not defined
```

Espace global, et espace local

Comme explicité au début du cours, le code de **définition** d'une fonction n'est **pas accessible** depuis l'extérieur. Cet espace propre à la fonction est appelé l'**espace local**. Le code en dehors de la fonction est appelé l'**espace global.**

L'espace global est disponible pour l'espace local d'une fonction, mais les objets définis dans l'espace local d'une fonction **ne sont pas accessibles** depuis l'espace global.

```
x = 15 # une variable x définie dans l'espace global
ma_liste = []

def truc():
# python va chercher x dans l'espace local et si il ne le trouve pas, il cherchera
x dans l'espace global
    return x

def alterer_une_liste(element):
# on peut même altérer les objets muables définis dans l'espace global depuis
l'espace local
    ma_liste.append(element)

truc()
alterer_une_liste(12)

print(x)        # 15
print(ma_liste) # [12]
```

>Les fonctions `globals()` et `locals()`

Ces fonctions permettent de dresser la liste des objets définis dans l'espace global et dans l'espace local.

```
from pprint import pprint

def faire_un_truc():
    variable_local = 12
    print("\nEspace local de l'intérieur de la fonction faire_un_truc: ")
    pprint(locals())

    print("\nEspace global vu depuis la fonction faire_un_truc: ")
    pprint(globals())

    return f"Je fais un truc avec {pi}."

pi = 3.14
faire_un_truc()

print("\nEspace global: ")
pprint(globals())

print("\nEspace local de l'espace global, c'est la même chose en fait: ")
pprint(locals())
```

On voit ici que `variable_local` n'est pas accessible depuis l'espace global, mais que l'intégralité de l'espace global depuis l'intérieur de la fonction `faire_un_truc()` est disponible.

Les fonctions `lambda`

Les fonctions `lambda` ou fonctions anonymes sont des fonctions n'ayant pas de noms et qui sont placées en paramètres d'autres fonctions. Elles sont alors appelées des `callback`.

Les fonctions anonymes sont présentes dans l'exemple dans la méthode `filter()` dans le chapitre sur les méthodes de base de la classe list, et dans le chapitre pour trier un dictionnaire.

Passage d'un paramètre par référence

Comme nous l'avons vu lorsque nous avons parlé des objets muables et immuables, certains objets sont passé par **valeur**, d'autres par **référence**. Les objets immuables comme les nombres sont passés par **valeur**, c'est à dire qu'une **copie** de l'objet est créé lorsqu'il sont appelés dans la fonction, et l'objet originel **n'est pas altéré**.

Au contraire, les listes qui sont des objets **muables**, vont être passé par **référence**, c'est à dire par **adresse mémoire** et seront altéré dans la fonction. On en a vu un exemple dans le code du titre "Espace global, et espace local" .

Récursivité

Définition

Une fonction est dite **récursive** lorsqu'elle **s'appelle elle même** à l'intérieur de sa propre définition.

```
def factorielle(nb):
    if nb == 1:
        return 1
    else:
        return nb * factorielle(nb-1)

print(factorielle(5)) # 120 --> 5! == 5 * 4 * 3 * 2 * 1
```

Dans cet exemple, la fonction va être appelée autant de fois que le nombre donné.

Exemples concret

La recherche dichotomique est un algorithme de recherche pour trouver la position d'un élément dans un tableau trié. Le principe est le suivant : comparer l'élément avec la valeur de la case au milieu du tableau ; si les valeurs sont égales, la tâche est accomplie, sinon on recommence dans la moitié du tableau pertinente.

```
liste = [1, 3, 5, 6, 7, 9, 12, 15, 18, 19, 23, 27, 30]

def binary_search(li, mini, maxi, elem):
    if maxi - mini == 1:
        if li[mini] == elem:
            return mini
        return maxi

    # On teste l'élément du milieu
    milieu = (maxi + mini) // 2
    if li[milieu] == elem:
        # On a eu de la chance, on est tombé sur l'élément du premier coup
        return milieu
    if li[milieu] > elem:
        # L'élément au milieu de la liste est > à ce qu'on recherche
        # Comme la liste est triée, notre élément est forcément avant le milieu
        return binary_search(li, mini, milieu - 1, elem)
    if li[milieu] < elem:
        # L'élément au milieu de la liste est < à ce qu'on recherche
        # Comme la liste est triée, notre élément est forcément après le milieu
        return binary_search(li, milieu + 1, maxi, elem)

print(binary_search(liste, 0, len(liste) - 1, 30))
```

📌 Cet algorithme mériterai des améliorations notamment une gestion de l'erreur si l'élément n'est même pas présent dans la liste.

Parcours de l'arborescence courante.

Ecrire une fonction récursive show_three qui affiche les noms des fichiers et qui se rappelle si un répertoire est rencontré en incrémentant l'indentation. Pour cette exercice, vous aurez besoin d'utiliser le slash (/) opérateur qui dans le cadre de pathlib permet de concaténer un chemin de manière compatible avec tous les os.

```
import os
from pathlib import Path

def show_tree(dir=".", indent=""):
    p = Path(dir)
    if not p.is_dir():
        print(f"{indent}{dir}")
        return  # Il faut ajouter un return ici pour arrêter l'exécution si ce
n'est pas un répertoire

    list_files_dirs = os.listdir(p)
    for fd in list_files_dirs:
        full_path = p / fd  # Construction du chemin complet
        print(f"{indent}{fd}")
        if full_path.is_dir():  # Utilisez full_path.is_dir() pour vérifier si
c'est un répertoire
            show_tree(full_path, indent + "-- ")  # On appelle la même fonction
avec le nouveau chemin

show_tree()
```

Exercice: "19. Les fonctions"