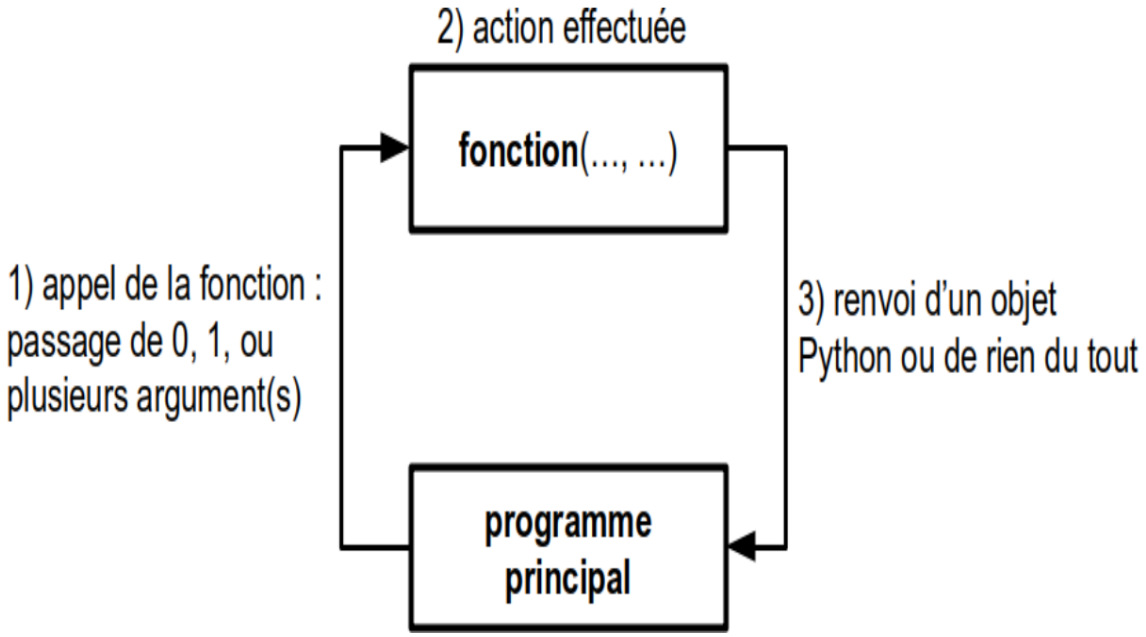
Plan du cours :

- ✓ Introduction
- ✓ Passage d'arguments
- ✓ L'instruction Return
- ✓ La portée des variables
- ✓ Exercices

1 Introduction

- En programmation, les fonctions sont très utiles pour réaliser plusieurs fois la même opération au sein d'un programme.
- Elles rendent également le code plus lisible et plus clair en le fractionnant en blocs logiques.
- Vous connaissez déjà certaines fonctions Python. Par exemple print() et input(), range() ou len()

1.1 Fonctionnement schématique d'une fonction



1.2 D'où viennent les fonctions????

Des fonctions Built-in	• Font partie intégrante de Python, comme print() et input() sont toujours disponibles sans ajouter aucun effort de la part du programmeur.
Des fonctions à partir des "**preinstalled modules**"	• Fonctions utiles mais pas autant utilisées que les **Built-in** fonctions, sont disponibles dans un certains nombres de modules préinstallés avec Python.
Nos propres fonctions	• Directement à partir de votre code, vous pouvez créer vos propres fonctions et les utiliser comme bon vous semble

La dernière possibilité est de créer les fonctions à l'intérieur des classes, nous allons voir cette possibilité dans les chapitres suivants.

1.3 Exemple d'appel de fonction

```
1  def message():
2      print("Enter a value: ")
3
4  message()
5  a = int(input())
6  message()
7  b = int(input())
8  message()
9  c = int(input())
10
```

2 Passage d'arguments

- Le nombre d'arguments que l'on peut passer à une fonction est variable.
- Nous avons vu ci-dessus des fonctions auxquelles on passait 0 ou 1 argument.
- Dans les chapitres précédents, vous avez rencontré des fonctions internes à Python qui prenaient au moins 2 arguments. Souvenez-vous par exemple de range(1, 10) ou encore range(1, 10, 2). Le nombre d'argument est donc laissé libre à l'initiative du programmeur qui développe une nouvelle fonction.
- Une particularité des fonctions en Python est que vous n'êtes pas obligé de préciser le type des arguments que vous lui passez, dès lors que les opérations que vous effectuez avec ces arguments sont valides. Python est en effet connu comme étant un langage au « typage dynamique », c'est-à-dire qu'il reconnaît pour vous le type des variables au moment de l'exécution.

2.1 Syntaxe :

```
def function(parameters) :
    instructiéons
```

Il faut retenir deux informations importantes à propos des paramètres des fonctions

- Les paramètres existent uniquement à l'intérieur des fonctions dans lesquelles ils sont définis, et bien sûr, l'argument (le paramètre) est défini entre les parenthèses après le nom de la fonction.
- Affecter une valeur au paramètre est réalisé au moment de l'appel de la fonction en spécifiant l'argument correspondant.

2.2 Exemple :

```
def somme(a,b):
    print("la somme des deux valeurs : " , a+b)

x=int(input("saisir une valeur: "))
y=int(input("saisir une valeur: "))
somme(x,y)
```

```
saisir une valeur: 15
saisir une valeur: 14
la somme des deux valeurs :  29
```

2.3 Positional parameter passing:

- Lorsqu'on définit la fonction def somme(a, b): les arguments a et b sont appelés arguments positionnels (en anglais positional arguments or paramters).
- Il est strictement obligatoire de les préciser lors de l'appel de la fonction. De plus, il est nécessaire de respecter le même ordre lors de l'appel que dans la définition de la fonction.
- Dans l'exemple ci-dessous, 14 correspondra à a et 15 correspondra à b. Finalement, tout dépendra de leur position, d'où leur qualification de positionnel.

2.4 Keyword argument passing :

- Python offre une autre façon de faire passer les arguments à une fonction, la signification de l'argument est dictée par son nom et non pas sa position, c'est ce qu'on appelle le passage par mot clé (keyword argument passing en anglais)

Exemple :

```
def introduction(firstName, lastName):
    print("Hello, my name is", firstName, lastName)

# appel de la fonction
x=input("saisir votre nom : ")
y=input("saisr votre prénom : ")
introduction(lastName=x,firstName=y)
introduction(firstName = "Harry", lastName = "Potter")
introduction(lastName = "Potter", firstName = "Harry")
```

```
saisir votre nom : BOUROUS
saisr votre prénom : Imane
Hello, my name is Imane BOUROUS
Hello, my name is Harry Potter
Hello, my name is Harry Potter
```

Mixing positional and keyword arguments:

- On peut mélanger entre les arguments positionnels et les arguments keywords, mais les arguments positionnels doivent toujours être placés avant les keywords arguments.

```
def somme(a, b, c):
    print(a, "+", b, "+", c, "=", a + b + c)

# appel de la fonction somme :

somme(1,2,3)    # 1 + 2 + 3 = 6
somme(c=1,a=2,b=3)  # 2 + 3 + 1 = 6
somme(3,c=1,b=2)   # 3 + 2 + 1 = 6
somme(3, a = 1, b = 2) #erreur multiple définition de a
somme(4,3,c=2) # 4 + 3 + 2 = 9
```

2.5 Valeur par défaut :

- Il est possible aussi d'affecter une valeur par défaut à un paramètre lors de la déclaration (valeur par défaut prédéfinie), cette valeur sera prise en considération dans le cas où l'argument n'est pas omis.

Exemple :

```
def somme(a,b=3):
    print(a," + ", b , " = " , a+b)

somme(5)    # 5  +  3  =  8
somme(a=2)   # 2  +  3  =  5
somme(a=1,b=2)   # 1  +  2  =  3
```

3 The return instruction :

Toute fonction peut éventuellement retourner une valeur, dans ce cas nous utiliserons l'instruction return, toutefois cette instruction peut avoir deux objectifs :

- En utilisant uniquement le mot clé return, sans spécifier de valeurs ou d'expressions, il permettra de terminer immédiatement l'exécution de la fonction, et le retour à la suite des instructions après l'appel de la fonction.
- La 2ème possibilité et d'utiliser le return avec une expression, nous utilisons la syntaxe suivante :

3.1 Syntaxe :

```
def function():
    return expression
```

Exemple : Retourner le maximum de deux nombres

```
def maximum(a,b):
    if a>b:
        return a
    else:
        return b

x=int(input("saisir un nombre : "))
y=int(input("saisir un nombre : "))
print("le maximum : " , maximum(x,y))
z=int(input("saisr un nombre : " ))
t=int(input("saisir un nombre: "))
#exploitation du return
m1=maximum(maximum(x,y),maximum(z,t))
print("le maximum des 4 nombres est : " , m1)
m2=maximum(maximum(maximum(x,y),z),t)
print("le maximum des 4 nombres est : " , m2)
```

3.2 Le mot clé None :

Lorsque l'instruction return est utilisée sans aucun argument, à l'intérieur d'une fonction, pour provoquer sa fermeture immédiate. La valeur retournée dans ce cas est l'objet None (objet particulier, correspondant à « rien »).

Attention : None ne doit être intégrée dans aucune expression, cela provoquera une erreur

Pour tester si une valeur est None nous utiliserons le test suivant ;

```
If valeur is None :
        ############
```

Passage d'une liste comme argument :

Il est possible de définir une fonction avec un paramètre liste, toutefois lors de l'appel de la fonction, il faut impérativement lui donner une liste et non pas un seul élément, ce qui risque de provoquer une erreur !

Exemple : calculer la somme des éléments d'une liste

```
def list_sum(lst):
    s = 0

    for elem in lst:
        s += elem

    return s

liste=[1,2,3,4,5,6]
print("la somme des éléments est : " , list_sum(liste))
# affichage : la somme des éléments est : 21
print("la somme des éléments est : " , list_sum(liste[0]))
# Erreur: int n'est pas itérable
```

Remarque : une fonction peut retourner une liste :

Exemple : construire dans une liste les diviseurs d'un nombre n

```
def diviseurs(n):
    liste=[]
    for i in range(1,n+1):
        if n%i==0:
            liste.append(i)
    return liste

n=int(input("saisir un nombre : "))
print("les diviseurs de " , n , " sont: ")
print(diviseurs(n))
```

Exemple : manipulation d'une liste contenant des chaines de caractères

```
def mafonction(liste1):
    for nom in liste1:
        print("Salam ", nom.capitalize())


mafonction(["imane","anas","aziz","amina"])
```

Exercice 1 : écrire une fonction qui prend comme paramètre une somme d'argent et qui la compose en billets et pièces, on se contentera des nombres entiers (billets et pièces disponibles : 200,100,50,20,10,5,2,1)

```
def somme_argent(n,liste1):
    liste2=[]
    for i in range(len(liste1)):
            s=n//liste1[i]
            liste2.append(s)
            n=n%liste1[i]
    return liste2

liste1=[200,100,50,20,10,5,2,1]
n=int(input("saisir une somme d'argent : "))
liste2=somme_argent(n,liste1)
for i in range(len(liste1)):
      print("nombre de " , liste1[i], " est : " , liste2[i])
```

4 La portée d'une variable :

Il est très important lorsque l'on manipule des fonctions de connaître la portée des variables (scope en anglais), c'est-à-dire savoir là où elles sont visibles. On a vu que les variables créées au sein d'une fonction ne sont pas visibles à l'extérieur de celle-ci car elles étaient locales à la fonction. Observez le code suivant :

```
def produit(a,b):
    m=a*b
    print("in fonction : la valeur de m est :" , m)
m=0
produit(5,8)
print("out fonction : la valeur de m est: " , m)
############################## Affichage
# in fonction : la valeur de m est : 40
# out fonction : la valeur de m est : 0
```

Lorsque Python exécute le code de la fonction, il connaît le contenu de la variable m. Par contre, de retour dans le programme principal, il ne la connaît plus, d'où l'affichage de la valeur de m avant l'exécution de la fonction. De même, une variable passée en argument est considérée comme locale lorsqu'on arrive dans la fonction.

Lorsqu'une variable est déclarée dans le programme principal, elle est visible dans celui-ci ainsi que dans toutes les fonctions.

```
def portee_variable():
    print("oui oui je connais la variable ", var)

var=3
portee_variable()
```

Toutefois, Python ne permet pas la modification d'une variable globale dans une fonction, L'erreur renvoyée montre que Python pense que var est une variable locale qui n'a pas été encore assignée. Si on veut vraiment modifier une variable globale dans une fonction, il faut utiliser le mot-clé global :

```
def portee_variable():
    print("oui oui je connais la variable ", var)
    var=4
    print("sa valeur devient : " , var)

var=3
portee_variable()
print("en dehors de la fonction : ", var)
```

Erreur affichée : UnboundLocalError: local variable 'var' referenced before assignment

Solution proposée:

```
def portee_variable():
    global var
    print("oui oui je connais la variable ", var)
    var=4
    print("sa valeur devient : " , var)

var=3
portee_variable()
print("en dehors de la fonction : ", var)
```

Résultat :

```
oui oui je connais la variable  3
sa valeur devient :  4
en dehors de la fonction :  4
```

Portée des listes : Soyez extrêmement attentifs avec les types modifiables (tels que les listes) car vous pouvez les changer au sein d'une fonction.

Exemple 1

```
def myFunction(myList1):
    print(myList1)
    myList1 = [0, 1]

myList2 = [2, 3]
myFunction(myList2)
print(myList2)
# affichage
# [2,3]
# [2,3]
```

Exemple 2 : la liste est modifiée dans ce cas :

```
def myFunction(myList1):
    print(myList1)
    del myList1[0]

myList2 = [2, 3]
myFunction(myList2)
print(myList2)
# affichage
# [2,3]
# [3]
```

Utilisation des fonctions récursives :

Une fonction récursive est une fonction qui s'appelle elle-même, toutefois, il faut faire attention à la condition d'arrêt sinon on risque d'exécuter la fonction à l'infinie

```
def factorielle(n):
    if n < 0:
        return None
    if n < 2:
        return 1
    return n * factorielle(n - 1)
a=int(input("saisir une valeur"))
print("la factorielle : ", factorielle(a))
```