Pense bête 3

Les fonctions

Une fonction est un ensemble d'instructions regroupées sous un **nom** et s'exécutant à la demande.

La définition d'une fonction est composée :

- du mot clé **def** suivi du **nom** de la fonction, de **parenthèses** vides si la fonction n'attend pas de paramètre ou entourant les **paramètres** de la fonction séparés par des virgules, et du caractère « deux points » ;
- d'une chaîne de documentation indentée comme le corps de la fonction ;
- du bloc d'instructions indenté par rapport à la ligne de définition, et qui constitue le corps de la fonction.
  Le bloc d'instructions est **obligatoire**. S'il est vide, on emploie l'instruction **pass**.

La documentation (facultative) est fortement conseillée.

```
#Définition de la fonction
def tirageDe():
    """ Retourne un nombre aléatoire compris entre 1 et 6 """
    from random import randint
    tirage = randint(1,6)
    return tirage
#Programme principal
for i in range(5):
  print(tirageDe()," ", end =" ")
```

Résultat :

2 5 2 4 3

Ici nous avons donc une fonction tirageDe, qui génère un nombre entier aléatoire et affecte ce nombre à la variable tirage.

La fonction renvoie la valeur de tirage.

Le programme principal, fait appel 5 fois à la fonction tirageDe et affiche la valeur renvoyée par la fonction.

Une fonction peut renvoyer deux valeurs.

```
#fonction de classement
def Classer(number1, number2):
  if number1 < number2:
    return number1, number2
  else:
    return number2, number1
#programme principal
n1, n2 = Classer(3, 2)
print("n1 est ", n1)
print("n2 est ", n2)
```

La fonction Classer reçoit 2 nombres, qu'elle classe par ordre croissant.

Elle renvoit les deux nombres classés.

Dans le programme principal, on récupère les deux valeurs renvoyées dans les deux variables n1 et n2.

Une fonction avec deux paramètres :

La fonction reçoit en paramètres deux valeurs donnant la valeur minimum et la valeur maximum de la variable aléatoire

```
#Définition de la fonction
def tirageDe2(min, max):
    """ Retourne un nombre aléatoire compris entre min et max """
    from random import randint
    tirage = randint(min,max)
    return tirage
#Programme principal
for i in range(5):
  print(tirageDe2(4,10)," ", end =" ")
```

Résultat :

6 7 9 4 9

Que ce passe t'il au moment de l'appel de la fonction avec des paramètres ?

Les **références** aux valeurs fournies au moment de l'appel (arguments), sont **copiées** dans l'ordre dans les paramètres de la fonction.

tirageDe2(4,10) → copie la référence à l'objet 4 dans le paramètre min, copie la référence à l'objet 10 dans le paramètre max.

Au moment de l'appel ce sont des **références à des objets** qui sont passées à la fonction.

Si l'argument est un nombre (4, 10 dans notre exemple) ou une chaîne de caractères, comme ceux-ci sont des objets **non mutables**, si des

modifications sont apportées au paramètre dans la fonction, ces modifications n'affectent pas l'argument.

Exemple :

```
#Définition de la fonction

def incremente(n):

    print("La valeur de n au début de la fonction est ",n)

    n = n + 1

    print("La valeur de n à la fin de la fonction est ",n)

#Programme principal

x = 1

print ("**Avant l'appel à incremente, la valeur de x est ",x)

incremente(x)

print ("**Après l'appel à incremente, la valeur de x est ",x)
```

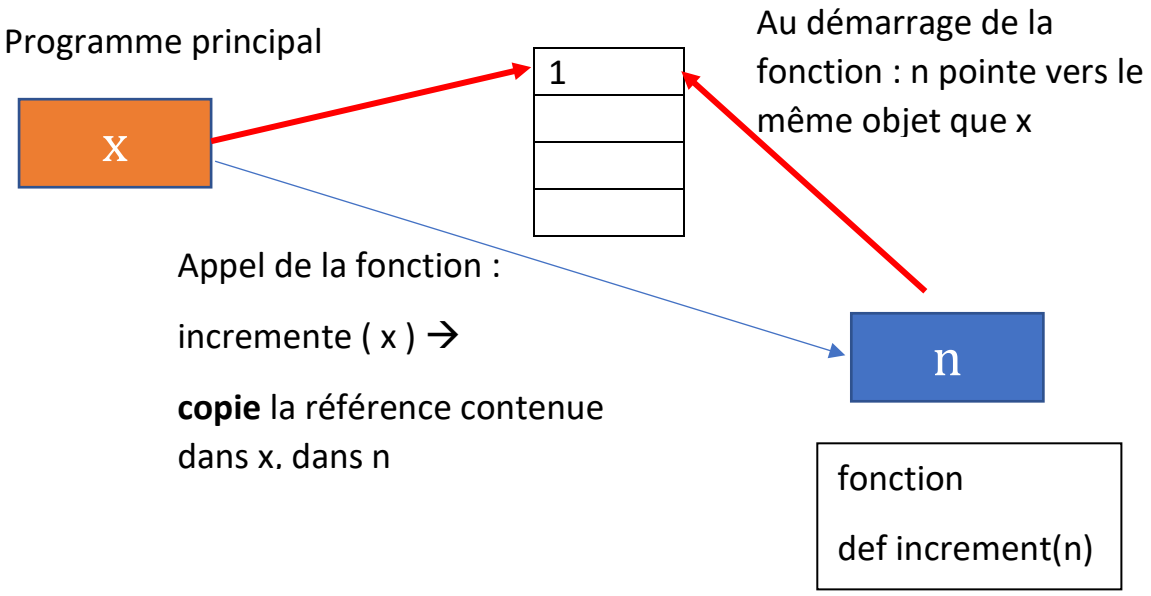
Résultat :

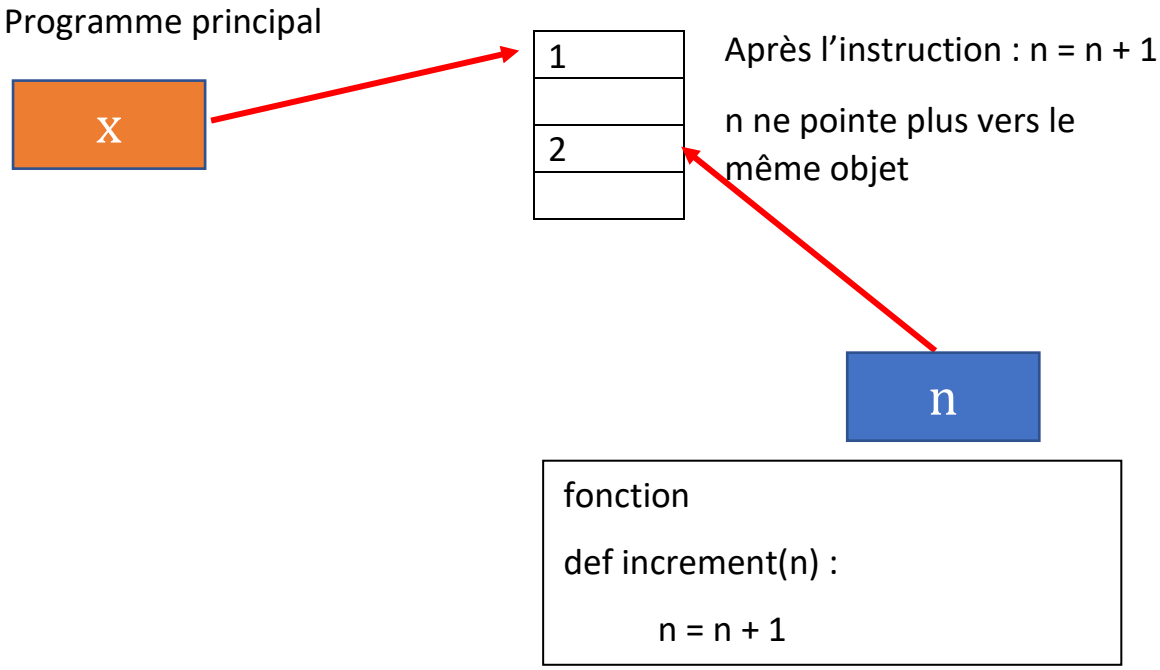
```
**Avant l'appel à incremente, la valeur de x est  1

La valeur de n au début de la fonction est  1

La valeur de n à la fin de la fonction est  2

**Après l'appel à incremente, la valeur de x est  1
```





Un objet entier n'étant pas mutable, lors de l'opération n = n + 1, python recherche la valeur de l'objet pointé par n, ajoute 1 à cette valeur, crée un nouvel objet pour y garer le résultat et place la référence à ce nouvel objet

dans n. n ne pointe plus vers la valeur passée en paramètre, mais vers la valeur résultat du calcul.

On peut mettre en évidence ce processus en utilisant l'instruction **id(variable)**

```
#Définition de la fonction
def incremente(n):
    print("Au démarrage de la fonction référence dans n :", id(n))
    n = n + 1
    print("A la fin de la fonction référence dans n ",id(n))
#Programme principal
x = 1
print ("**Avant l'appel à incremente, référence dans x ",id(x))
incremente(x)
print ("**Après l'appel à incremente, référence dans x ",id(x))
```

Résultat :

****Avant l'appel** à incremente, référence dans **x** : 1864741088

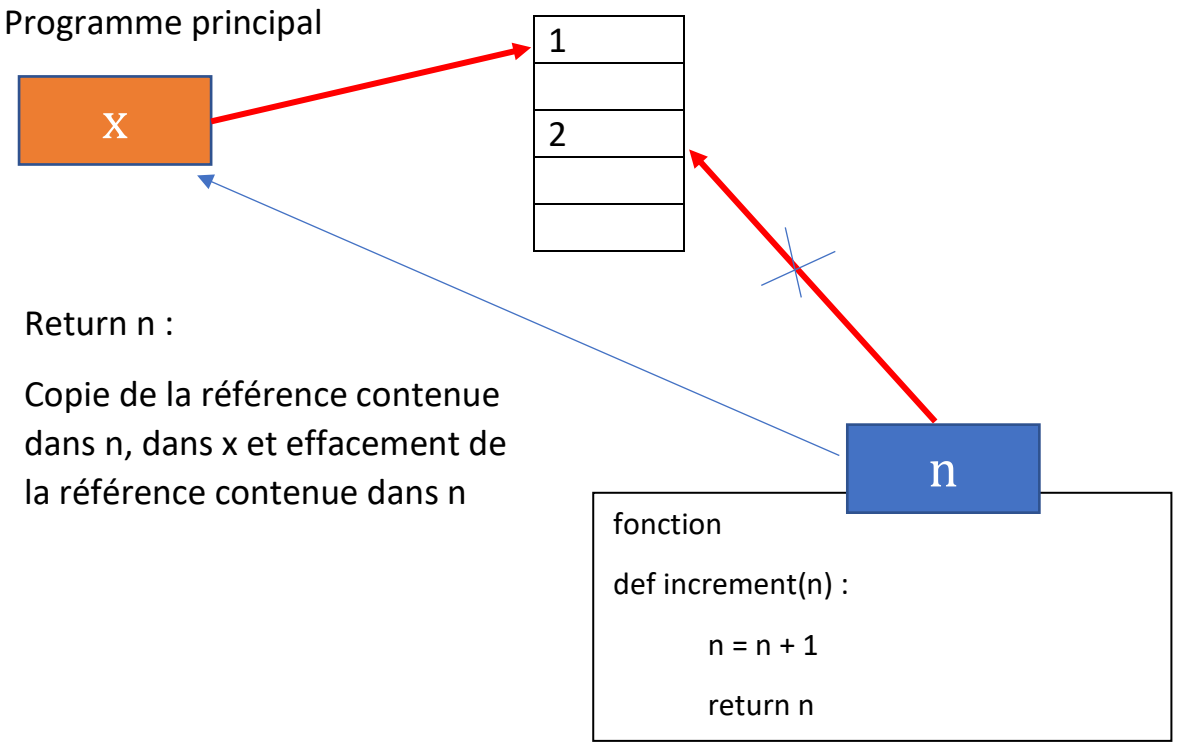
Au démarrage de la fonction référence dans **n** : 1864741088

A la fin de la fonction référence dans **n** : 1864741120

****Après l'appel** à incremente, référence dans **x** : 1864741088

Et si je récupère dans x la valeur de n à la fin de la fonction en ajoutant une instruction return dans la fonction ?

```
def incremente(n)
    n = n + 1
    return n
#Programme principal
x = 1
x = incremente(x)
```



Regardons les valeurs des références des variables x et n, avant et après le rerour de la fonction :

```
#Définition de la fonction
def incremente(n):
    print("Au démarrage de la fonction référence dans n :", id(n))
    n = n + 1
    print("A la fin de la fonction référence dans n ",id(n))
    return n
#Programme principal
x = 1
print ("**Avant l'appel à incremente, référence dans x ",id(x))
x  = incremente(x)
print ("**Après l'appel
```

Résultat :

**Avant l'appel à incremente, référence dans x 1864741088

Au démarrage de la fonction référence dans n : 1864741088

A la fin de la fonction référence dans n 1864741120

**Après l'appel à incremente, référence dans x 1864741120

Après le retour de la fonction, n n'est plus accéssible.

Passage en paramètre d'une liste

Rappel : Une liste est un objet **mutable**.

```
#Définition de la fonction
def test(maliste):
    print ("Au début de la fonction : maliste = ", maliste)
    maliste.append(4) #on ajoute un élément à la liste
    print ("A la fin de la fonction : maliste = ", maliste)
#Programme principal
l1 = [1,2,3]
print ("**Dans le programme principal, avant appel à la fonction: l1 = ", l1)
test(l1)
print ("**Dans le programme principal, après appel à la fonction: l1 = ", l1)
```

Résultat :

```
**Dans le programme principal, avant appel à la fonction: l1 =  [1, 2, 3]
Au début de la fonction : maliste =  [1, 2, 3]
A la fin de la fonction : maliste =  [1, 2, 3, 4]
**Dans le programme principal, après appel à la fonction: l1 =  [1, 2, 3, 4]
```

Nous voyons que lors de l'exécution de l'instruction **maliste.append(4)** dans la fonction, python ne crée pas une nouvelle liste après y avoir ajouté un élement.

La référence dans le paramètre maliste, reçue au moment de l'appel n'est pas modifiée par l'opération append.

Les 2 variables **maliste** et **l1** conservent toutes les deux les mêmes références et pointent donc toutes les deux vers la même liste, avant, après, pendant l'exécution de la fonction.