Type de document	Cours et exercices	Classe	1re	Durée	4h	Date	15/12/2022
Thème et contenu(s)	Langages et programmation – Fonctions						
Capacités attendues	Savoir créer et appeler une fonction dans son programme						
Prérequis	Déclaration de variables, types de base, listes, conditions, boucles						
Description	Résumé du cours sur les fonctions						

I) Définition

Une fonction en informatique (y compris en Python donc) est une **portion de code nommée** et **réutilisable** qui **effectue une tâche** spécifique à chaque fois qu'elle est **appelée**. Elle peut s'utiliser avec ou sans paramètres et peut renvoyer un résultat (pas l'afficher, le renvoyer)

La chose que vous devez garder à l'esprit même si les exercices que nous faisons sont parfois trop simples pour que l'on s'en rende bien compte est que l'intérêt d'une fonction est de **réutiliser** le code et de faciliter sa **maintenance** (changements). Si on a un code de 10 lignes présent 5 fois dans notre programme, au lieu de le copier-coller, on le met dans une fonction et on appelle cette fonction à la place.

II) Syntaxe basique

Il faut distinguer deux choses : la **définition** d'une fonction, et son **appel**.
Sa définition doit toujours intervenir avant son appel (c'est pareil pour les variables : on doit définir une variable avant de pouvoir l'utiliser). Une fonction qui n'est jamais appelée n'est jamais exécutée.

II - a) Fonction sans paramètres

Pour définir une fonction, on utilise la syntaxe : `def nomFonction():`
C'est un exemple simple de fonction sans aucun paramètre. On écrit le code de la fonction sur toutes les lignes suivantes (il faut bien sûr penser à **indenter**, comme pour les boucles et les structures conditionnelles).
Notez le mot clef `def`, suivi d'un espace, suivi du nom de la fonction, suivi de parenthèses ouvrantes et fermantes et des :
Pour appeler la fonction, il suffit d'aller dans le code (donc à l'extérieur de la fonction) et d'écrire son nom suivi des parenthèses. Voici 4 exemples commentés de l'exécution de fonctions sans paramètres qu'on définit et qu'on appelle ensuite.

```
1  #définition
2  def disCoucou():
3      print("COUCOU !")
4
5  #appel
6  coucou()
```

```
COUCOU
>
```

```
1  #définition
2  def perroquet():
3      k=input("dis un truc : ")
4      print(k)
5
6  #appel
7  perroquet()
```

```
dis un truc : j'aime Python !
j'aime Python !
>
```

```python
#affiche les multiples de 9
#entre 400 (inclus) et 500 (exclu)

#définition
def multiplesDe9():
    print("je suis une fonction")
    print("qui affiche les multiples de 9")
    for i in range(400,500):
        if i%9==0:
            print(i)

#appel
multiplesDe9()
```

```
je suis une fonction
qui affiche les multiples de 9
405
414
423
432
441
450
459
468
477
486
495
```

```python
def asciiHeart(): #début définition
    print("          ******       *******")
    print("        **********   **********")
    print("      ************ ************")
    print("     ******************************")
    print("     ******************************")
    print("     ******************************")
    print("      ****************************")
    print("       **************************")
    print("         **********************")
    print("           ******************")
    print("             **************")
    print("              ************")
    print("                 *******")
    print("                   ***")
    print("                    *")
    #fin de la définition

#appel
asciiHeart()
```

fichier.py

PROBLÈMES 8 SORTIE CONSOLE DE DÉBOGAGE TERMINAL JUPY

```
PS C:\Users\Utilisateur\Desktop\Programmation\Python>  c:;
er/../..\debugpy\launcher' '22894' '--' 'c:\Users\Utilisate
          ******       ******
        **********   **********
      ************ ************
     ******************************
     ******************************
     ******************************
      ****************************
       **************************
          ******************
            **************
             ************
                *******
                  ***
                   *
PS C:\Users\Utilisateur\Desktop\Programmation\Python>
```

console

II - a) Fonction avec paramètres

Lorsqu'on désire que notre fonction utilise des valeurs pour faire des calculs, on va spécifier ces valeurs sous forme de paramètres dans les parenthèses. S'il y en a plusieurs on les sépare par des virgules. L'intérêt d'un paramètre est donc d'être utilisé dans la fonction. **En aucun cas il ne doit avoir de valeur : on le traite comme une variable !**
Comme en mathématiques, on ne définit jamais une fonction comme ceci en Python : ~~f(3) = 2x+5~~
C'est faux et ça n'a pas de sens d'écrire ça !
Voici des exemples ainsi que leur exécution :

```
1  #définition
2  def carre(cote):
3      print("je dessine un carré de côté", cote)
4      for i in range(cote):
5          for j in range(cote):
6              print(" * ",end="") #end="" sert à ne
                   pas sauter de ligne
7          print("")
8
9  #appels
10 carre(1)
11 carre(2)
12 carre(4)
13 carre(10)
14
15
```

```
je dessine un carré de côté 1
 *
je dessine un carré de côté 2
 *  *
 *  *
je dessine un carré de côté 4
 *  *  *  *
 *  *  *  *
 *  *  *  *
 *  *  *  *
je dessine un carré de côté 10
 *  *  *  *  *  *  *  *  *  *
 *  *  *  *  *  *  *  *  *  *
 *  *  *  *  *  *  *  *  *  *
 *  *  *  *  *  *  *  *  *  *
 *  *  *  *  *  *  *  *  *  *
 *  *  *  *  *  *  *  *  *  *
 *  *  *  *  *  *  *  *  *  *
 *  *  *  *  *  *  *  *  *  *
 *  *  *  *  *  *  *  *  *  *
 *  *  *  *  *  *  *  *  *  *
```

```
1  #définition
2  def pairs(a,b):
3      for i in range(a,b+1):
4          if i%2 == 0:
5              print(i,"est pair")
6
7  #appels
8  pairs(1,5)
9  pairs(1456,1488)
10
11
```

```
2 est pair
4 est pair
1456 est pair
1458 est pair
1460 est pair
1462 est pair
1464 est pair
1466 est pair
1468 est pair
1470 est pair
1472 est pair
1474 est pair
1476 est pair
1478 est pair
1480 est pair
1482 est pair
1484 est pair
1486 est pair
1488 est pair
```

Lorsqu'on passe des paramètres à une fonction lors de son appel, ces paramètres sont soit des valeurs littérales (types de base : entiers, flottants, booléens, chaînes de caractères) comme on le voit à la ligne 10 de la capture suivante, soit des variables contenant des valeurs littérales, comme le montrent les lignes 8 et 9.

```
1  #fonction qui renvoie l'inverse d'une chaîne
       de caractères
2  def inverse(s):
3      resultat = "" #chaîne vide
4      for i in range(len(s)):
5          resultat = s[i]+resultat
               #concaténation
6      print(resultat)
7
8  a = "salut"
9  inverse(a)
10 inverse("j'aime bien coder")
```

```
tulas
redoc neib emia'j
>
```

Enfin, il n'est pas obligatoire de nommer les paramètres de la même manière en-dehors de la fonction, et dans la fonction. Sur la capture ci-dessus, dans la fonction, la chaîne passée en entrée se nomme s.
Lors de l'appel ligne 9, le paramètre d'entrée est a. L'interpréteur Python se charge, lors de l'appel de la fonction, de faire correspondre les paramètres. Donc lors de l'appel de la fonction `inverse()`, le paramètre s prendra la valeur de a.

Attention : si une fonction prend plusieurs paramètres, il faut veiller à les lui donner dans le bon ordre

```
1  #n est un entier
2  #s est une chaîne de caractères
3  def affiche(n,s):
4      for i in range(n):
5          print(s)
6
7  i = 4
8  texte = "j'aime répéter des trucs"
9  affiche(i, texte)
```

```
j'aime répéter des trucs
j'aime répéter des trucs
j'aime répéter des trucs
j'aime répéter des trucs
>
```

III) Syntaxe avec return

III - a) Présentation

Il est important, pour comprendre l'utilité du return, de comprendre que toutes les variables créées dans une fonction n'existent par défaut que dans cette fonction. Voici un exemple simple :

```
1  def addition(a,b):
2      c = a+b
3      print(c)
4
5  addition(5,8)
6  print(c)
```

```
13
Traceback (most recent call last):
  File "<string>", line 6, in <module>
NameError: name 'c' is not defined
>
```

La variable `c`, calculée à partir des paramètres a et b, est connue dans la fonction. On peut même l'afficher (ligne 3). Dès qu'on « sort » de la fonction, c n'est plus connue (la ligne 6 provoque donc

une erreur). **On dit que c est une variable locale, ou encore que la portée de la variable c est locale.**
Si on veut récupérer le résultat de l'exécution d'une fonction, par exemple pour le traiter, le réutiliser, on doit dire à cette fonction de nous renvoyer ce résultat. On va utiliser pour cela le mot-clef `return`. On ne peut retourner qu'une valeur (et non pas une suite de valeurs). Si on veut renvoyer plusieurs choses on peut les mettre dans une liste, par exemple.

Exemple avec la fonction précédente :

```
1 def addition(a,b):
2       c = a+b
3       return c
4
5  addition(5,8)
```

Le seul souci est que lorsqu'on ne stocke pas ce que nous retourne une fonction, ce retour est « perdu » à jamais. Il faut le récupérer : pour cela on utilise une variable quelconque :

```
1 def addition(a,b):
2       c = a+b
3       return c
4
5  z=addition(5,8)
6  print("ma fonction a renvoyé",z)
```

```
ma fonction a renvoyé 13
>
```

Ici c'est dans la variable z qu'on stocke le résultat du retour de notre fonction.

Important : ne confondez pas ce qu'affiche une fonction à l'écran et ce qu'elle retourne. Une fonction peut très bien afficher des choses si elle possède des `print()` dans son code, et ne rien renvoyer. Elle peut aussi renvoyer quelque chose et ne rien afficher. Ou elle peut renvoyer et afficher, ou même ne rien renvoyer et ne rien afficher… Ci-dessus, la fonction addition() n'affiche rien et renvoie quelque chose. C'est nous qui, plus tard, à l'extérieur de cette fonction, choisissons d'afficher la valeur qu'elle a renvoyée, mais nous aurions pu tout aussi bien ne pas le faire. Donc en aucun cas cette fonction n'affiche quoi que ce soit. **Pour qu'une fonction renvoie quelque chose il faut obligatoirement utiliser le mot-clef return**

III - b) Cas des conditions

Quand on a des conditions dans une fonction, et qu'on veut qu'elle renvoie quelque chose, on doit s'assurer qu'elle renvoie quelque chose à chaque fois. C'est ce qu'on appelle une bonne pratique : les bonnes pratiques en programmation sont un ensemble de règles qui permettent de coder efficacement (en évitant les bugs) et de manière claire. Voici un exemple de ce qu'il ne faut pas faire :

```
1 def pair(nb):
2     if nb%2==0:
3         return True
4
5 res1 = pair(8)
6 res2 = pair(5)
7 print(res1)
8 print(res2)
```

```
True
None
>
```

Comme vous le voyez, le cas où le nombre passé en paramètre est impair n'est pas traité : rien n'est prévu dans la fonction pour ça. Par conséquent, elle ne retourne rien dans le cas où le nombre passé en paramètre est impair et la ligne 8 affichera « None » ce qui signifie que la fonction n'a rien renvoyé. Les bonnes pratiques nous imposent de faire plutôt quelque chose comme ça :

```
1  def pair(nb):
2      if nb%2==0:
3          return True
4      else:
5          return False
6
7  res1 = pair(8)
8  res2 = pair(5)
9  print(res1)
10 print(res2)
```

```
True
False
>
```

Il faut donc toujours se poser la question suivante lorsqu'on crée une fonction et qu'on veut qu'elle retourne quelque chose : est-ce qu'elle renvoie bien quelque chose dans tous les cas ?

III - c) Le return, ce Terminator

Appeler return dans une fonction provoque l'arrêt de son exécution, il sert de **terminateur**. On peut donc utiliser, même si ce n'est pas conseillé, return sans rien ensuite, juste pour dire à la fonction qu'elle doit s'arrêter :

```
1  def terminator():
2      print("je veux tes vêtements, tes bottes et ta moto")
3      print("Tu dois faire ce que je dis ?")
4      print("C'est un des paramètres de ma mission")
5      return
6      print("Mets-toi sur un pied !")
7      print("Hasta la vista, Baby")
8
9  terminator()
```

```
je veux tes vêtements, tes bottes et ta moto
Tu dois faire ce que je dis ?
C'est un des paramètres de ma mission
>
```

Ici, la fonction (appelée à la ligne 9) n'exécutera jamais les lignes 6 et 7. Il peut même y avoir un autre return, cela ne change rien, il ne sera jamais exécuté :

```
1  def terminator():
2      print("je veux tes vêtements, tes bottes et ta moto")
3      print("Tu dois faire ce que je dis ?")
4      print("C'est un des paramètres de ma mission")
5      return
6      print("Mets-toi sur un pied !")
7      print("Hasta la vista, Baby")
8      return
9
10 terminator()
```

```
je veux tes vêtements, tes bottes et ta moto
Tu dois faire ce que je dis ?
C'est un des paramètres de ma mission
>
```

Important : on ne peut exécuter qu'une seule fois return dans une fonction. Si le code de la fonction comporte plusieurs return, comme ci-dessus, ou comme dans la fonction pair(), un seul d'entre eux s'exécutera

IV) Exemples

IV - a) Fonction qui renvoie l'inverse d'une chaîne de caractères

```
def inverse(s):
    r= ''
    for i in range(len(s)):
        r = s[i] + r #on concatène avant !
    return r

def inverse2(s):
    r = ''
    for i in range(len(s)-1,-1,-1): #revoyez la notion de range
        r = r + s[i]
    return r

r1 = inverse("Hasta la vista, baby")
r2 = inverse("KoT-KiT")
print(r1)
print(r2)
```

```
ybab ,atsiv al atsaH
TiK-ToK
>
```

Remarques :

- On ne peut pas modifier une chaîne de caractères comme on modifie une liste, il faut donc recréer une chaîne de caractères vide et la remplir, puis à la fin la renvoyer
- La version 2 parcourt la chaîne de sa fin : `len(s)-1` (pour ne pas avoir un out of range) jusqu'à -1 exclu (donc jusqu'à l'indice 0), de -1 en -1. Revoyez la syntaxe du range en vous entraînant à faire des boucles.
- La version 1 parcourt la chaîne dans l'ordre mais elle ajoute chaque élément au début de la nouvelle chaîne

IV - b) Fonction qui teste si un mot est un palindrome

On va appeler la fonction `inverse()` à l'intérieur de notre fonction `palindrome()` : on est dans le cas où des fonctions s'appellent entre elles.

```
def palindrome(s):
    if s==inverse(s):
        return True
    else:
        return False

r1 = palindrome("leon a suce ses ecus a noel")
print(r1)
```

```
True
>
```

IV - c) Fonction qui affiches toutes les combinaisons possibles entre deux listes (de tailles différentes ou égales, peu importe)

```
def toutesLesCombinaisons(l1,l2):
    for e in l1:
        for f in l2:
            print(e,f)

jus = ["pomme", "orange", "ananas"]
alcools = ["rhum", "vodka", "tequila"]
print("liste de cocktails :")
toutesLesCombinaisons(alcools, jus)
```

```
liste de cocktails :
rhum pomme
rhum orange
rhum ananas
vodka pomme
vodka orange
vodka ananas
tequila pomme
tequila orange
tequila ananas
```