Le JEU pour REVISER les bases du PYTHON

Récapitulatif des cours présents dans le jeu • **Chapitre 4**

Table des matières

Les fonctions

Les **fonctions** sont l'une des notions les plus importantes en programmation (pour changer…).

Une fonction est une suite d'instructions (comme les conditions et les boucles) que l'on peut appeler avec un nom (un peu à la manière d'une variable).

Elles sont plusieurs informations :

- Un **nom**
- Un/des **paramètres**
- Une/des **instructions**
- Une/des **valeurs de retour**
- Un/des **appels**

1. Les fonctions et les maths

Dans un premier temps, on peut comparer les fonctions en programmation aux fonctions mathématiques.

Une fonction mathématique à elle aussi :

- Un nom
- Un/des paramètres
- Une/des instructions
- Une/des valeurs de retour
- Un/des appels

$$f(x) = 2x+1$$

$f(3) \rightarrow 2 * 3 + 1 = 7$

$f(0) \rightarrow 2 * 0 + 1 = 1$

Voici ça traduction en python :

```
# Fonction
def f(x):
    return 2 * x + 1

# Programme principal
a = f(3)
b = f(0)

print(a)    # 7
print(b)    # 1
```

def définie la fonction f avec un paramètre x.

return indique le résultat retourné par f.

Les variables a et b appellent la fonction f pour x = 3 et x = 0 et récupèrent le résultat.

3 et 0 sont des arguments.

Exemples de fonction mathématique :

```
# Fonction
def ma_fonction(x, y):
    return 2 * x + y / 3


# Programme principal
a = ma_fonction(3, 6)
b = ma_fonction(0, 12)

print(a)          # Affiche 8.0
print(b)          # Affiche 4.0
```

2. La structure

Comment indiqué précédemment, la structure d'une fonction est la suivante :

- Un nom => Un nom unique (voir Chapitre 1 sur les variables)
- Un/des paramètres => Il peut y avoir aucun paramètre
- Une/des instructions => Au minimum une ligne
- Une/des valeurs de retour
  => Pas de return : la fonction renvoie « None »
  => Un seul return : à mettre à la fin de la fonction
  => Plusieurs return : en fonction des conditions/boucles
  => Retourner plusieurs valeurs : utilisé une liste/tuple/dico
- Un/des appels
  => Dans le programme principal
  => Le programme principal se stoppe pour effectuer les instructions de la fonction, puis continu
  => Les valeurs passées dans l'appel de la fonction sont appelées des « arguments »

La structure à utiliser :

```
# Fonction
def   le_nom_de_la_fonction(parametre_1, parametre_2, ... ):
      # instruction 1
      # instruction 2
      # ...
      return valeur_a_retournee

# Programme principal
appel_1 = le_nom_de_la_fonction(argument_1, argument_2, ... )
appel_2 = le_nom_de_la_fonction(argument_a, argument_b, ... )
```

Exemples :

```
# Fonction
def fusion_chaine(ch1, ch2):
    fu = ch1 + " et " + ch2
    return fu


# Programme principal
a = fusion_chaine("une", "deux")
b = fusion_chaine("chat", "chien")

print(a)      # Affiche "une et deux"
print(b)      # Affiche "chat et chien"
```

Code couleurs :

```
def fusion_chaine(ch1, ch2):
    fu = ch1 + " et " + ch2
    return fu

a = fusion_chaine("une", "deux")
b = fusion_chaine("chat", "chien")
```

- 1 nom
- 2 paramètres
- 1 instruction
- 1 valeur de retour
- 2 appels avec 2 arguments

3. L'utilisation

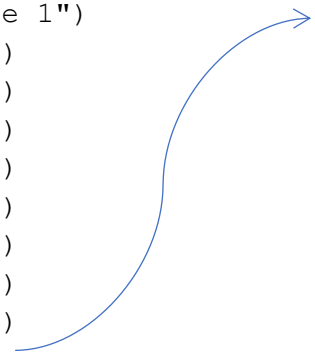
Une **fonction** peut avoir plusieurs utilités :

- Optimiser un programme
- Ordonner un programme
- Créer des modules

A. Optimiser un programme

Prenons le programme python suivant :

```
print("Partie 1")
print(1, "a")
print(2, "a")
print(3, "a")
print(4, "a")
print(5, "a")
print(6, "a")
print(7, "a")
print(8, "a")
```



```
print("Partie 2")
print(1, "b")
print(2, "b")
print(3, "b")
print(4, "b")
print(5, "b")
print(6, "b")
print(7, "b")
print(8, "b")
print(9, "b")
print(10, "b")
```

Il est possible d'optimiser ce programme à l'aide de boucle **for** :

```
print("Partie 1")
for i in range(1, 9):
    print(i, "a")

print("Partie 2")
for i in range(1, 11):
    print(i, "b")
```

C'est mieux ! Cependant, on remarque qu'il y a des ressemblances entre la première et la seconde partie du programme.

C'est là qu'intervienne les fonctions !

```
# Fonction
def fonction_boucle(lettre, nombre):
    for i in range(1, nombre+1):
        print(i, lettre)

# Programme principal
print("Partie 1")
fonction_boucle("a", 8)

print("Partie 2")
fonction_boucle("b", 10)
```

Les instructions de la fonction **fonction_boucle** sont celles qui étaient identiques dans la précédente version (la boucle **for**).

```
def fonction_boucle(lettre, nombre):
    for i in range(1, nombre+1):
        print(i, lettre)
```

Les valeurs qui changeaient sont remplacées par des **variables**.
Les valeurs de ces variables sont indiquées lors des **appels de la fonction**.

```
print("Partie 1")
fonction_boucle("a", 8)

print("Partie 2")
fonction_boucle("b", 10)
```

/! *Dans une fonction, il faut éviter d'utiliser "print" (voir la suite du cours)*

B. Ordonner un programme

Supposons un programme de plusieurs milliers de ligne.

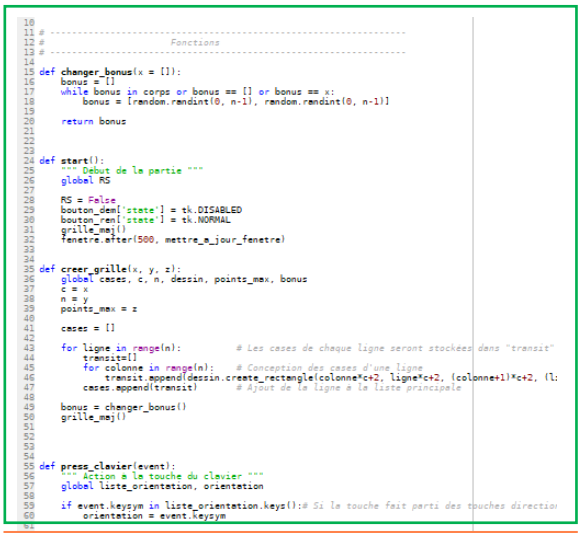
Pas facile de s'y retrouver !

Les fonctions peuvent permettre de créer des sous-parties au programme pour mieux s'y retrouver.

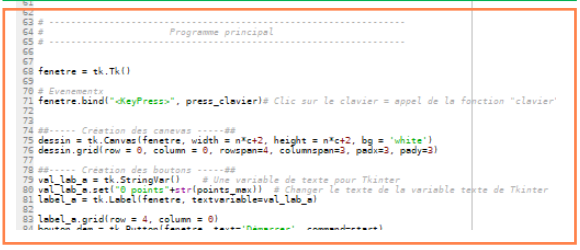
Conseils

- Il est généralement préférable d'écrire les **fonctions** en premiers, puis d'écrire le **programme principal** (appels des fonctions).

- Si le programme est important, il est aussi possible de mettre le programme principal dans un **fichier** et les fonctions dans d'**autres fichiers**.



```
# ------------------------------------------------------------
#                          Fonctions
# ------------------------------------------------------------

def changer_bonus(x = []):
    bonus = []
    while bonus in corps or bonus == [] or bonus == x:
        bonus = [random.randint(0, n-1), random.randint(0, n-1)]

    return bonus


def start():
    """ Debut de la partie """
    global RS

    RS = False
    bouton_dem['state'] = tk.DISABLED
    bouton_ren['state'] = tk.NORMAL
    grille_maj()
    fenetre.after(500, mettre_a_jour_fenetre)


def creer_grille(x, y, z):
    global cases, c, n, dessin, points_max, bonus
    c = x
    n = y
    points_max = z

    cases = []

    for ligne in range(n):          # Les cases de chaque ligne seront stockées dans "transit"
        transit=[]
        for colonne in range(n):    # Conception des cases d'une ligne
            transit.append(dessin.create_rectangle(colonne*c+2, ligne*c+2, (colonne+1)*c+2, (l
        cases.append(transit)       # Ajout de la ligne à la liste principale

    bonus = changer_bonus()
    grille_maj()



def press_clavier(event):
    """ Action à la touche du clavier """
    global liste_orientation, orientation

    if event.keysym in liste_orientation.keys():# Si la touche fait parti des touches directio
        orientation = event.keysym
```



```
# ------------------------------------------------------------
#                       Programme principal
# ------------------------------------------------------------


fenetre = tk.Tk()

# Evenements
fenetre.bind("<KeyPress>", press_clavier)# Clic sur le clavier = appel de la fonction "clavier"


##----- Création des canevas -----##
dessin = tk.Canvas(fenetre, width = n*c+2, height = n*c+2, bg = 'white')
dessin.grid(row = 0, column = 0, rowspan=4, columnspan=3, padx=3, pady=3)

##----- Création des boutons -----##
val_lab_a = tk.StringVar()    # Une variable de texte pour Tkinter
val_lab_a.set("0 points"+str(points_max))  # Changer le texte de la variable texte de Tkinter
label_a = tk.Label(fenetre, textvariable=val_lab_a)

label_a.grid(row = 4, column = 0)
```

C. Créer des modules

Les modules, ou bibliothèques, sont des fichiers contenant des fonctions que l'on peut utiliser dans plusieurs programmes en même temps.

```
import tkinter as tk
import random
from math import sqrt
```

Python possède de nombreuses bibliothèques préinstallées (math, random, tkinter, etc.).

Il est possible d'en télécharger des préexistantes ou d'en créer soi-même.

Pour plus de détails sur les modules, RDV en classe de Terminale NSI :)

/!\ VIGILENCE : Les effets de bord

Il est généralement déconseillé d'utiliser **input** et **print** dans une fonction.

Il est préférable de les utiliser dans le programme principal, et d'utiliser des **paramètres** et des **return** pour les interactions entre le programme principal et les fonctions.

Pourquoi ?

print, **input**, **global** et d'autres opérations d'entrée/sortie interagissent avec l'extérieur sans laisser de trace dans le programme principal.

On appelle cela un **effet de bord**. C'est un peu un « effet secondaire » de la fonction.

C'est généralement déconseillé d'en avoir pour pouvoir débugger plus efficacement un programme.

Une bonne fonction **ne doit pas** :

- Avoir d'autres entrée/sortie que les paramètres et les return
- Modifier une variable du programme principal (voir la suite)
- Retourner un résultat différent pour un même paramètre ou groupe de paramètres (ex : le résultat de x*5 est toujours la même quand x = 2)

Attention à la modification des listes, tuples et dictionnaires :

Lorsque l'on passe une valeur en argument d'une fonction et qu'elle est modifié dans cette fonction, elle n'est pas modifiée dans le programme principal.

Exemple :

```
def fonction(x):
    x = 3
    return x

val = 2
fonction(val)
print(val)      # Affiche 2

re = fonction(val)
print(re)       # Affiche 3
```

Il y a une **exception** pour les listes, les tuples et les dictionnaires.

Exemple :

```
def fonction(li):
      li.append("b")

ma_liste = [1, 2, 3]
fonction(ma_liste)
print(ma_liste)       # Affiche [1, 2, 3, "b"]

fonction(ma_liste)
print(ma_liste)       # Affiche [1, 2, 3, "b", "b"]
```

Il est conseillé de faire une **copie** de la liste/tuple/dictionnaire dans la fonction, puis de retourner la structure de données (c'est plus simple pour suivre les actions du programme).

Vous pouvez utiliser la fonction **copy** (liste/tuple/dico) ou **deepcopy** (fonctionne aussi sur les matrices : liste/tuple/dico dans les listes/tuples/dico) de la bibliothèque **copy**.

Exemple :

```
from copy import deepcopy

def fonction(li):
      nouvelle_li = deepcopy(li)
      nouvelle_li.append("b")
      return nouvelle_li

ma_liste = [1, 2, 3]
fonction(ma_liste)
print(ma_liste)       # Affiche [1, 2, 3]

re = fonction(ma_liste)
print(re)             # Affiche [1, 2, 3, "b"]
```

Précision : Parfois, nous n'avons pas le choix d'avoir des effets de bord (tout dépend du programme). L'important est de les réduire le plus possible.

En savoir plus sur les fonctions

- https://python.doctor/page-apprendre-creer-fonction-en-python
- https://fr.wikipedia.org/wiki/Effet_de_bord_(informatique)
- https://pyspc.readthedocs.io/fr/latest/05-bases/03-fonctions.html
- Ou demande à ton/ta prof de NSI :)