LYCÉE Jules Haag CPGE PT / PTSI
Votre avenir avec un temps d'avance

TP d'Informatique

Les fonctions dans python

TP3

Partie du programme :

1. Programme du premier semestre

1.1 Rappels des notions de programmation en langage python vues au lycée

L'objectif de ce TP est de transposer des algorithmes de résolution d'un problème posé, en script écrit dans le langage de développement python.

Ressources **:**

- **Cours 4 : Les fonctions**

Rappels :

```
def nom_fonction(liste de paramètres):
    bloc d'instructions
```

Il existe **3 types de fonctions sous python** :

- **Fonction qui fait toujours la même chose**

Exemple :

```
def compteur():                          # définition de la fonction
    i = 0
    while i < 3:
        print(i,end=" ")
        i = i + 1

print("\nbonjour")
compteur()                               # 1er appel de la fonction
print()
compteur()                               # 2ieme appel de la fonction
```

- **Fonction avec passage de paramètres**

Exemple :

```
# définition de la fonction paramétrée
def compteur_complet(start, stop, step):
    i = start
    while i < stop:
        print(i, end=" ")
        i = i + step

 # 1er appel de la fonction (start, stop, step) sont des copies de (1, 7, 2)
compteur_complet(1, 7, 2)
print()

# 2ieme appel de la fonction (start, stop, step) sont des copies de (2, 6, 1)
compteur_complet(2, 6, 1)
```

Un argument peut être une valeur, une chaine de caractère, une autre fonction.
Les paramètres reçus par la fonction sont appelés ***arguments****.*
Lors de l'appel de la fonction, le paramètre peut être passé sous la forme de valeurs ou sous la forme de noms de variables.

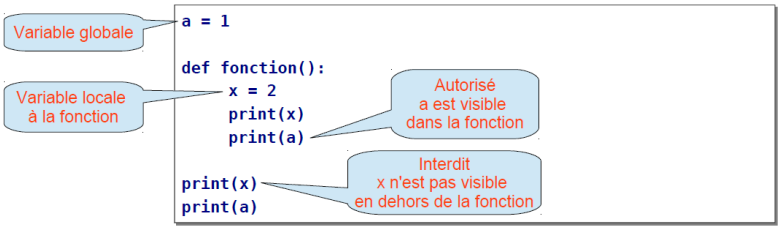
- **Fonctions qui retournent une valeur**

Exemples :

```
def cube(w):          # définition de la fonction paramétrée
    return w**3       # la fonction « renverra » une valeur bien déterminée
                      # exploitable par le programme


c=cube(3)         # le programme récupère la valeur renvoyée par
                  # la fonction en l'affectant à une variable
print(c)
```

- *La fonction « renvoie » toujours une valeur bien déterminée exploitable par le programme appelant.*
- *Si l'instruction « return » n'est pas utilisée, ou si elle est sans argument, la fonction renvoie un objet vide : « None ».*
- *« return » peut être suivi du nom d'une variable ou d'une expression.*



PROGRAMME 1 : SIGNE ET VALEUR ABSOLUE

1. Créer un script val_absolue.py
2. Écrire une fonction **signe(x)** ayant un seul argument qui renvoie « 1 » si cet argument est strictement positif, « -1 » s'il est strictement négatif, « 0» s'il est nul.
3. À l'aide de **signe(x)**, écrire une fonction **valAbs(x)** qui renvoie la valeur absolue de son argument.

PROGRAMME 2 : RESOLUTION D'UNE EQUATION DU 2ND DEGRE

On souhaite utiliser les résultats du cours « ***Python – Les fonctions*** » sur « la fonction **Equation()** qui renvoie la liste des racines d'une équation du second degré ».

1. Créer un script resol_degre2.py

- **Utilisation de la fonction**
  2. Écrire un programme qui appelle cette fonction en passant 3 paramètres **a, b et c** prenant la valeur (3, 2 et -1), récupère le résultat dans un tableau **racines** et qui affiche, suivant le nombre de racines (0, 1 ou 2), le nombre de racines, suivi de leurs valeurs s'il y a lieu.
- **Ajout d'une fonction :**
  3. Au lieu d'analyser puis d'afficher directement le résultat, ajouter une seconde fonction **analyseResultat(racines)** qui retourne une chaine de caractère qui contient l'analyse du résultat (comme précédemment). Simplifier la première fonction en conséquence.
  4. Tester en affichant le résultat de la fonction **analyseResultat(racines)**
- **Modifier le programme pour qu'il propose la saisie des paramètres a, b et c**

PROGRAMME 3 : MESURES HYGROMETRIQUES

On souhaite faire l'acquisition d'une suite de mesures quotidiennes de l'hygrométrie de l'air exprimée en % pour en calculer la moyenne. Pour cela, on souhaite écrire un programme qui :

a. Demande d'abord le nombre de mesures à acquérir.
b. Demande ensuite la saisie des valeurs par une boucle. Les valeurs saisies sont vérifiées à chaque fois par une fonction **verification(val)**. Cette fonction a pour seul paramètre la valeur à vérifier, et renvoie le booleen **True** si la valeur saisie est correcte (c'est à dire un nombre homogène à un pourcentage) et **False** sinon. Si la valeur saisie n'est pas correcte, le programme la redemande.
c. Affiche enfin la moyenne des mesures à l'aide d'une fonction **moyenne(somme, nb)** qui prend en premier argument une somme de valeurs et en second le nombre de valeurs utilisées dans le calcul de cette somme.

Application :

1. Créer un script hygrometrie.py
2. Écrire le corps du programme avec les appels des fonctions **verification** et **moyenne** qui répond à ce problème.
3. Écrire ensuite le code des fonctions **verification(val)** et **moyenne(somme, nb)**.

PROGRAMME 4 : ATTENTE

Lors d'une mise à jour d'une application sur téléphone ou sur PC, un affichage montre le temps restant. On demande un script simple pour réaliser cette tâche.

Vous importerez la bibliothèque « time » et pourrez utiliser dans ce module les fonctions :

- time.time() : renvoie le nombre de secondes écoulées depuis l'époque sous forme de nombre à virgule flottante. L'époque est le point de départ du temps. Le 1er janvier de cette année, à 0 heure, le « temps écoulé depuis l'époque » est égal à zéro. En général, l'époque est 1970. Pour savoir quelle est l'époque, regardez time.gmtime(0).
- time.clock() : renvoie le nombre de secondes écoulées depuis le premier appel de cette fonction sous forme de nombre à virgule flottante.
- time.sleep(seconds) : suspend l'exécution de la tâche actuelle pendant le nombre de secondes indiqué.

1. Créer un script attente.py
2. Écrire une fonction **timer()** qui :
   - Affichera le texte « téléchargement commencé ».
   - Affichera le « temps restant » en secondes, sans virgule, toutes les secondes, pour atteindre une valeur de compte à rebours de 10 secondes.
   - Affichera le texte « téléchargement terminé » à la fin des 10 secondes.
3. Appeler la fonction et vérifier.
4. Améliorer votre script en déclarant en variable globale la valeur de compte à rebours. Tester.

Conseils :

- *Utiliser des variables locales dans la fonction (stockage de l'heure de début de l'appel de la fonction, etc.)*
- *Utiliser une variable booléenne pour sortir d'une boucle while.*

1. Exercices supplémentaires :

Exercice 1 : Approximations du nombre reel π

On propose deux méthodes d'approximation du nombre réel π.

1. Méthode des trapèzes

   On sait que $\pi = 4\int_0^1 f(t)\,dt$, où $f(t) = \sqrt{1-t^2}$.

   On admet que la suite $(T_n)_{n\geq 1}$ définie ci-dessous converge vers $\int_0^1 f(t)\,dt$:

   $$T_n = \frac{1}{n}\sum_{i=0}^{n-1} \frac{f\left(\frac{i}{n}\right) + f\left(\frac{i+1}{n}\right)}{2}.$$

   Ecrire une fonction **trapeze** (d'argument un entier n) donnant une valeur approchée de π à l'aide de la suite (T_n).

2. Méthode Brent-Salamin

   Cette seconde méthode, de 1975, repose sur les travaux de R.Brent et E.Salamin.

   Soit $a_0 = 1,\ b_0 = \frac{1}{\sqrt{2}},\ s_0 = \frac{1}{2}$ et pour $n \geq 0$:

   $$a_{n+1} = \frac{a_n + b_n}{2},\ b_{n+1} = \sqrt{a_n b_n},\ c_{n+1} = a_{n+1}^2 - b_{n+1}^2,\ s_{n+1} = s_n - 2^{n+1}c_n.$$

   Alors la suite (p_n) définie par $p_n = \frac{2a_n^2}{s_n}$ converge vers π.

   Ecrire une fonction **brent-salamin** (d'argument un entier n) donnant une valeur approchée de π à l'aide de la suite (p_n).

   Remarque : On peut montrer qu'on double le nombre de bonnes décimales de π à chaque itération.

Exercice 2 : Adressage IP

L'adresse IP d'un poste informatique dans un réseau **TCP/IP** est composée de 4 octets. On la représente usuellement sous la forme de 4 chiffres décimaux (de 0 à 255) séparés par des points (ex : **172.30.32.1**)

Une adresse IP est en fait composée de **2 parties** : l'adresse réseau, et l'adresse d'hôte.
Le masque de sous réseaux permet de distinguer ces 2 parties de l'adresse IP. Il est de la forme **255.255.0.0**.
Le (ou les) octet(s) de ce masque, qui valent **0,** définisse(nt) la partie d'adresse hôte (dans cet exemple, les 2 derniers octets : **32.1**).
L'adresse réseau est donc composée des 2 premiers octets (ceux dont l'octet du masque est à **255**), suivi d'une adresse d'hôte à **0** (dans l'exemple : **172.30.0.0**).
L'adresse de diffusion est définie par l'adresse réseau, avec des octets de l'adresse d'hôte à **255** (dans l'exemple : **172.30.255.255**)

Pour résumer :

Adresse IP	**172**	**30**	**32**	**1**
Masque	**255**	**255**	**0**	**0**
Adresse réseau	**172**	**30**	**0**	**0**
Adresse de Diffusion	**172**	**30**	**255**	**255**

Pour simplifier l'étude, on ne prendra en compte que les réseaux de classe A (masque **255.0.0.0**), classe B (masque **255.255.0.0**) et classe C (masque **255.255.255.0**).

- **Version 1 : masque de sous réseau prédéfini**

1. Réaliser un programme qui, à partir de la saisie successive des 4 octets (en décimal) d'une adresse IP, affiche l'adresse IP, le masque, l'adresse réseau, ainsi que l'adresse de diffusion (le masque étant saisi « en dur » dans le code).
   On commencera par déclarer 4 listes (adIP, masque, adR, adD) de 4 valeurs chacun, destinés à contenir les valeurs décimales des : adresse IP, masque de sous réseau, adresse réseau et adresse de diffusion).
   Ces 4 listes (sauf le masque) seront préalablement initialisées à [0,0,0,0].

2. Tester votre programme avec les 3 adresses suivantes **10.0.0.5/255.0.0.0**, **172.16.1.23/255.255.0.0** et **192.168.1.27/255.255.255.0**.

- **Version 2 : en utilisant des fonctions**

3. Modifier le programme précédent en créant :
   - Une fonction **saisieAdIP()** qui retourne une liste contenant les 4 valeurs
   - Une fonction **calculeAdRes(IP, msq)** qui retourne, à partir de l'adresse IP et du masque , une liste contenant l'adresse réseau
   - Une fonction **calculeAdDiff(IP, msq)** qui retourne, à partir de l'adresse IP et du masque , une liste contenant l'adresse de diffusion
   - Une fonction **afficheAdresses(IP, msq, res, diff)** qui se charge de l'affichage des 4 adresses