FONCTIONS

1. Principe et généralités

En programmation, pour réaliser plusieurs fois la même opération au sein d'un programme, on utilise des fonctions qui sont des éléments structurants de base de Python (de manière plus générale de tout langage informatique).

Elles offrent différents avantages :

- **Évitent la répétition** : on peut « isoler » une portion de code qui se répète lors de l'exécution en séquence d'un script ;
- **Mettent en relief les données et les résultats** : entrées et sorties de la fonction
- **Permettent la réutilisation :** permission de réutiliser les sorties de la fonction
- **Décomposent une tâche complexe en tâches plus simples**

Elles rendent donc le script (d'un algorithme relativement complexe) plus lisible et plus clair en le fractionnant en blocs.

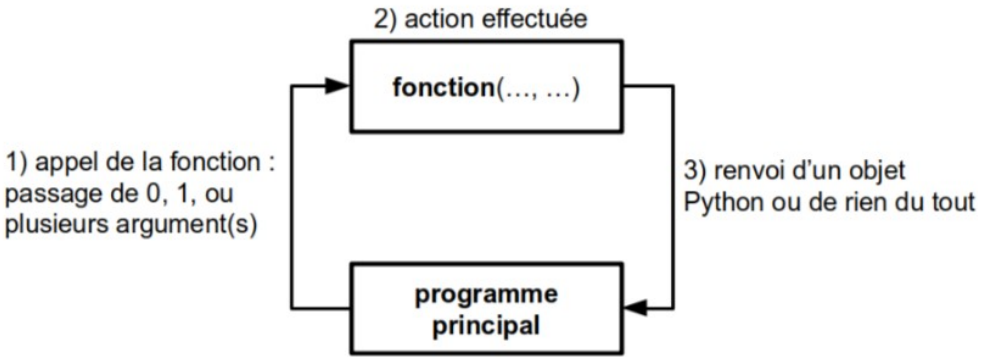
On connaît déjà certaines fonctions Python.
Par exemple la fonction **len()** qui renvoie la longueur d'une liste ou d'une chaîne de caractères.

Entrée :
```
In [1]: len([0,1,2])
```

1. On appelle **len()** en lui passant une liste une variable qui dans ce contexte sera appelée argument (ici [0, 1, 2])
2. la fonction **len()** calcule la longueur de cette liste
3. elle vous renvoie un entier égal à cette longueur

Sortie :
```
Out[1]: 3
```

De manière générale, on peut se représenter une fonction comme une sorte de « boîte noire » que l'on peut schématiser de la manière suivante :



Du point de vue d'un programmeur, une fonction est une portion de code effectuant une suite d'instructions bien particulière.

Chaque fonction effectue en général une tâche **unique et précise**. Si cela se complique, il est plus judicieux d'écrire plusieurs fonctions (qui peuvent éventuellement s'appeler les unes les autres). Cette **modularité** améliore la qualité générale et la lisibilité du code.

Pour finir sur les généralités, on a utilisé dans la figure ci-dessus le terme **programme principal** pour désigner l'endroit depuis lequel on appelle une fonction.
Le programme principal désigne le code qui est exécuté lorsqu'on lance le script Python, c'est-à-dire toute la suite d'instructions en dehors des fonctions. En général, dans un script Python, on écrit d'abord les fonctions puis le programme principal.

2. Définir une fonction

Syntaxe à suivre

```
def maFonction(arg):
  suite d'instructions
  suite d'instructions
  ...
  return resultat
```

Une fonction est donc composée :

- d'une ligne d'entête commencée par le mot-clef **def** et terminée par deux points, constituée du nom de la fonction et de ses arguments (variables auxquelles va s'appliquer la fonction)
- d'un bloc d'instructions indenté par rapport à la ligne d'entête
- d'une instruction **return** qui renvoie le résultat (aussi indentée) qui peut être récupérée dans une variable

A propos de l'instruction return

Saisir dans l'éditeur et notez ce que vous obtenez dans le shell

```
def carre_1(x):
    y=x**2
    return y


In [20]: res=carre(3)

In [21]: print(res)
```

```
def carre_2(x):
    y=x**2
    print(y)

In [16]: res=carre(3)

In [17]: print(res)
```

Conclusion :

Remarque une fonction ne prend pas nécessairement un argument

L'exécution du script suivant :

```
def hello():
    print('bonjour')
```

nous donne :

```
In [27]: hello()
bonjour
```

3. Un premier exemple

Retour sur l'algorithme de dichotomie du TP4.

On souhaiterait modifier l'algorithme suivant afin qu'il soit utilisable pour n'importe quelle fonction.

```
1  n=int(input("précision voulue : 10^-"))
2  h=10**-n              # amplitude de l'intervalle
3  a=1
4  b=2
5
6  while b-a>=h:
7      m=(a+b)/2
8      fa=a**3+a-3       # calculs des images de a et m par f
9      fm=m**3+m-3
10     print(fa,fm)
11
12     if fa*fm<0:
13         b=m
14
15     else:
16         a=m
17
18 print(a,"< x0 <",b)
```

Pour cela, on le modifie de la façon suivante :

```
1  def f(x):                          } définition de la fonction
2      y=x**3+x-3
3      return y
4
5  n=int(input("précision voulue : 10^-"))
6  h=10**-n
7  a=1
8  b=2
9
10 while b-a>=h:                      <- appel de la fonction
11     m=(a+b)/2                      } programme principal
12     if f(a)*f(m)<0:
13         b=m
14     else:
15         a=m
16
17 print(a,"< x0 <",b)
```

Ainsi le programme principal n'est pas modifié.

On remarque qu'il est aussi plus lisible.

4. Variables locales et variables globales

Lorsqu'on manipule des fonctions, il est essentiel de bien comprendre comment se comportent les variables. Une variable est dite **locale** lorsqu'elle est créée dans une fonction. Elle n'existera et ne sera visible que lors de l'exécution de ladite fonction.

Une variable est dite **globale** lorsqu'elle est créée dans le programme principal. Elle sera visible partout dans le programme.

Exemple Sur le site *http://www.pythontutor.com* qui permet de visualiser l'état des variables au fur et à mesure de l'exécution d'un code Python saisir le programme suivant :

```
1  #definition de la fonction
2  def carre(x):
3      y=x**2
4      return y
5
6  #programme principal
7
8  z=5
9  res=carre(z)
10 print(res)
```

Cliquez sur Visualize Execution, puis sur next << First < Prev Next > Last >> pour suivre l'exécution du programme pas à pas.

Commentaires à lire en même temps que le suivi de l'exécution du programme

Étape 1 Python est prêt à lire la première ligne de code.

Étape 2 Python met en mémoire la fonction carre(). Il ne l'exécute pas!
La fonction est mise dans un espace de la mémoire nommé *Global frame*, il s'agit de l'espace du programme principal. Dans cet espace, seront stockées toutes les variables *globales* créées dans le programme. Python est maintenant prêt à exécuter le programme principal.

Étape 3 Python lit et met en mémoire la variable z.
Celle-ci étant créée dans le programme principal, il s'agira d'une **variable *globale***.
Ainsi, elle sera également stockée dans le *Global frame*.

Étape 4 La fonction carre() est appelée et on lui passe en argument l'entier z.
La fonction s'exécute et un nouveau cadre est créé dans lequel *Python Tutor* va indiquer toutes les variables *locales* à la fonction.
La variable passée en argument, qui s'appelle x dans la fonction, est créée en tant que **variable *locale***.
On remarquera aussi que les variables *globales* situées dans le *Global frame* sont toujours là.

Étape 5 Python est maintenant prêt à exécuter chaque ligne de code de la fonction.

Étape 6 La variable y est créée dans la fonction.
Celle-ci est donc stockée en tant que **variable *locale*** à la fonction.

Étape 7 Python s'apprête à renvoyer la variable *locale* y au programme principal. *Python Tutor* nous indique le contenu de la valeur renvoyée.

Étape 8 Python quitte la fonction et la valeur renvoyée par celle-ci est affectée à la variable *globale résultat.*
Notez bien que lorsque Python quitte la fonction, **l'espace des variables alloué à la fonction est détruit**.
Ainsi, toutes les variables créées dans la fonction n'existent plus. On comprend pourquoi elles portent le nom de *locales* puisqu'elles n'existent que lorsque la fonction est exécutée.

Étape 9 Python affiche le contenu de la variable résultat et l'exécution est terminée.

5. Exercices

(a) Les coefficients binomiaux $\binom{n}{k}$

On souhaite écrire une fonction qui renvoie les coefficients binomiaux $\binom{n}{k}$.

On appelle coefficient binomial $\binom{n}{k}$ le nombre : $\frac{n!}{(n-k)!k!}$ où : $n!=n\times(n-1)\times(n-2)\times\ldots\times2\times1$

i. Définir la fonction **fact** pour qu'elle renvoie $n!$, pour tout entier naturel n.

ii. Compléter la fonction **coefficient** qui renvoie $\binom{n}{k}$ sous forme d'entier lorsque n est un entier naturel et k un entier naturel inférieur ou égal à n.

(b) Statistiques

i. Créez une fonction gen_distrib() qui prend comme argument trois entiers : debut, fin et n. La fonction renverra une liste de n floats aléatoires entre debut et fin. Pour générer un nombre aléatoire dans un intervalle donné, utilisez la fonction uniform() du module random dont voici quelques exemple d'utilisation :

```
>>> import random
>>> random.uniform(1, 10)
8.199672607202174
>>> random.uniform(1, 10)
2.607528561528022
>>> random.uniform(1, 10)
9.000404025130946
```

Avec la fonction random.uniform(), les bornes passées en argument sont incluses, c'est-à-dire qu'ici, le nombre aléatoire renvoyé est dans l'intervalle [1, 10].

ii. Créez une autre fonction calc_stat() qui prend en argument une liste de floats et qui renvoie une liste de trois éléments contenant respectivement le minimum, le maximum et la moyenne de la liste.

iii. Dans le programme principal, générez n listes aléatoires de 100 floats compris entre 0 et 100 et affichez le minimum (min()), le maximum (max()) et la moyenne pour chacune d'entre elles. La moyenne pourra être calculée avec les fonctions sum() et len().

iv. Pour chacune des 5 listes, affichez les statistiques (min, max, et moyenne) avec deux chiffres après la virgule :

```
debut : 1
fin : 10
nombre de listes : 5
ligne 1 : [4.592469765477313, 9.92022401091235, 7.009042613032324]
ligne 2 : [3.310273010720705, 9.510099525568052, 5.8973584203081835]
ligne 3 : [1.9623469783276213, 5.553715675391499, 3.8188012947133023]
ligne 4 : [2.3658141304603726, 5.879113657796415, 3.9559673856602293]
ligne 5 : [1.2699434211864649, 8.831170065635764, 6.2669010320602965]
```

Les écarts sur les statistiques entre les différentes listes sont-ils importants ?
Relancez votre script avec des listes de 1000 éléments, puis 10 000 éléments. Les écarts changent-ils quand le nombre d'éléments par liste augmente ?

(c) **Distance**

i. Créez une fonction **`calc_distance_carre()`** qui calcule la distance euclidienne entre deux points au carré.
On rappelle que la distance euclidienne d entre deux points A et B de coordonnées cartésiennes respectives (xA;yA) et (xB;yB) se calcule comme suit :
$d = \sqrt{(xB - xA)^2 + (yB - yA)^2}$

ii. Testez votre fonction sur les 2 points A(0 ;0) et B(1,1). Trouvez-vous bien 2 ?

iii. Dans le programme principal, vous utiliserez votre fonction **`calc_distance_carre()`**, pour déterminer si un triangle ABC est rectangle ou pas.
Vous respecterez l'écran de sortie suivant :

```
saisir l'ascisse du point A: 1
saisir l'ordonnée du point A: 0
saisir l'ascisse du point B: 5
saisir l'ordonnée du point B: 0
saisir l'ascisse du point C: 5
saisir l'ordonnée du point C: 3
ABC est rectangle en B
```

(d) suites

On considère la suite (u_n) définie par : $\begin{cases} u_0 = 1 \\ u_{n+1} = 3\,u_n + 5 \end{cases}$ *pour tout* $n \in \mathbb{N}$

i. Proposer une fonction F1 qui prend pour argument un entier naturel n et retourne u_n.
On n'utilisera pas de listes dans la fonction.

ii. Proposer une fonction F2 qui prend pour argument un entier naturel n et retourne la liste $L = [u_0, u_1, \ldots u_n]$.

(e) Recherche d'un indice dans une liste

On s'intéresse à des mesures de niveau de la surface de la mer.

On appelle horodate un ensemble (fini) des mesures réalisées sur une période de 20 minutes à une fréquence d'échantillonnage de 2 Hz. Les informations de niveau de surface (déplacement vertical en m) sont stockées dans une liste de flottants liste_niveaux. On suppose qu'aucun des éléments de cette liste n'est égal à la moyenne.

i. Proposer une fonction **`moyenne`** prenant en argument une liste non vide `liste_niveaux` et retournant sa valeur moyenne.

ii. Proposer une fonction **`ind_premier_pzd`** prenant en argument une liste non vide `liste_niveaux` et retournant, s'il existe, l'indice du premier élément de la liste tel que cet élément soit supérieur à la moyenne et l'élément suivant soit inférieur à la moyenne.
Cette fonction devra retourner « —1 » si aucun élément vérifiant cette condition n'existe.

iii. Proposer une fonction retournant l'indice `i` du dernier élément de la liste tel que cet élément soit supérieur à la moyenne et l'élément suivant soit inférieur à la moyenne.
Cette fonction devra retourner « -2 » si aucun élément vérifiant cette condition n'existe.

COMPLEMENTS

Le Triangle de Pascal

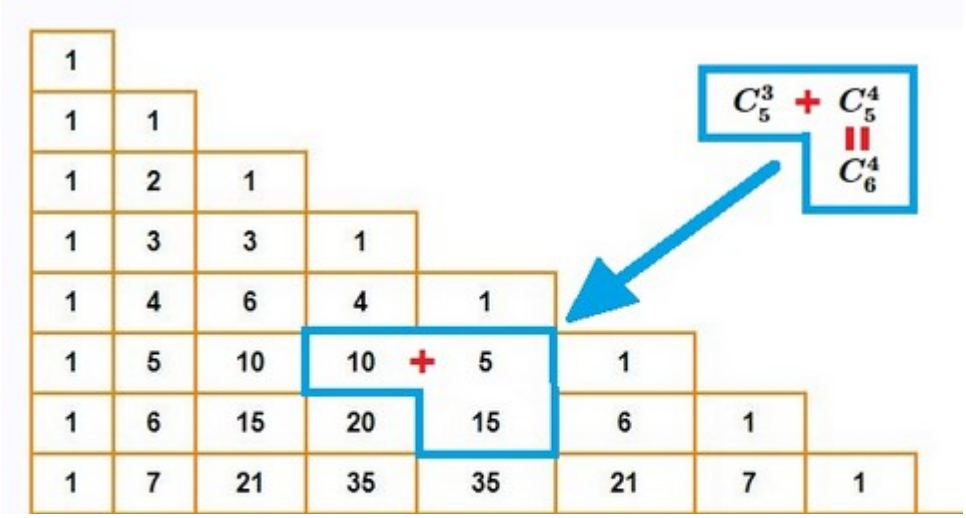
Le triangle de Pascal est en quelque sorte un schéma permettant de calculer les coefficients binomiaux de proche en proche de manière simple.

Sa construction est basé sur la propriété suivante :

pour tous entiers naturels n et k tel que $k \leqslant n$: $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-2}$

remarque on peut noter $\binom{n}{k}$ par C_n^k

Illustration



Écrire un fonction **triangle_Pascal** prenant en argument un entier naturel n et renvoyant les n+1 premières lignes du triangle de Pascal sous forme de liste de listes.

Ainsi, **triangle_Pascal(3)** doit renvoyer : **[[1],[1,1],[1,2,1],[1,3,3,1]]**.

Algorithme de tri

On considère une liste définie par extension par l'instruction **maliste = [1,7,3,5,1,2,8]**.

Expérimentation

1. Que fait l'instruction **maliste.remove(7)** sur la variable **maliste** ?

2. Que fait l'instruction **maliste.remove(1)** sur la variable **maliste** ?

Application

1. Proposer une fonction **tri_selection** prend en argument une liste de nombres et renvoie une nouvelle liste contenant les mêmes éléments triés par ordre croissant.
   On pourra utilier la fonction **min** qui renvoie le minimum d'une liste de nombres
2. Modifier la fonction **tri_selection** renvoie la liste triée par ordre décroissant.