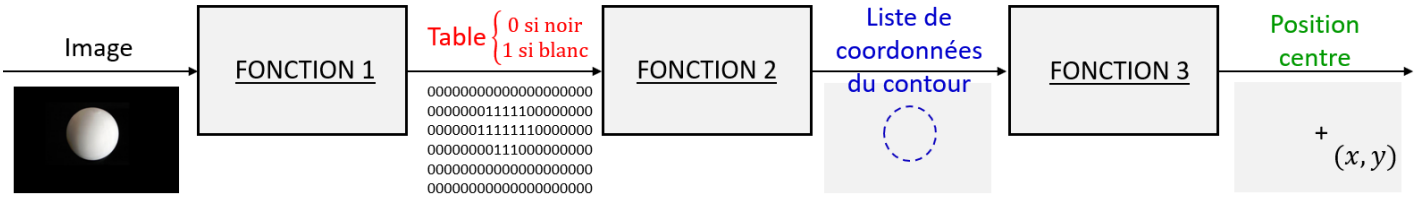
Informatique Pour Tous

Cours 2 – Fonctions et portée des variables

A / Quelques intérêts des fonctions – évaluation aux concours

Lors du développement d'un projet informatique, plusieurs personnes peuvent être amenées à coder en parallèle sur le même projet. Une équipe va donc se répartir le travail en décomposant une tâche complexe en tâches unitaires, un **script principal** (souvent appelé ***main***, terme anglais) étant chargé d'exécuter dans l'ordre chacun des scripts unitaires. Cette vision séquentielle est cependant très simpliste. En pratique, les tâches ne sont souvent pas totalement dissociables. Prenons par exemple un algorithme de détection du centre d'une balle blanche sur fond noir :



On décompose ici la tâche en 3 fonctions :

- Numériser l'image et appliquer un seuil, pour donner une matrice de 0 (pixel sombre) et de 1 (pixel clair)

  Argument de *f1* : Image **Renvoi** de *f1* : Matrice

- Détecter un contour fermé sur une matrice, et renvoyer les coordonnées des pixels de ce contour

  Argument de *f2* : Matrice **Renvoi** de *f2* : Liste de coordonnées

- Calculer le barycentre d'une liste de points

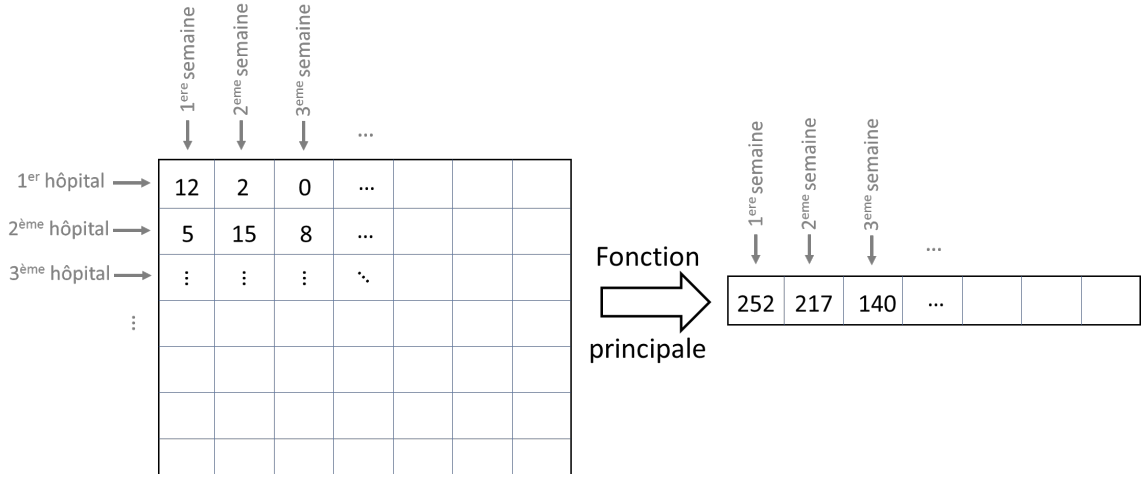
  Argument de *f3* : Liste de coordonnées **Renvoi** de *f3* : Coordonnées (x,y) du centre

Pour représenter le schéma ci-dessus, on écrira en informatique :

Position_centre = *f*3 (Liste_coordonnées) = *f3* (*f2* (Matrice)) = *f3* (*f2* (*f1* (Image)))

On voit donc que la tâche complexe (prenant en argument une image, et renvoyant la position du centre de la balle) a été décomposée en 3 fonctions, mais ces fonctions peuvent être écrites par 3 personnes différentes, du moment qu'elles se mettent d'accord sur le format d'entrée (**argument**), et le format de sortie (ce que **renvoie** la fonction).

Entre autres, il est intéressant de définir la notion de fonction **auxiliaire**. Imaginons qu'on dispose d'une base de données qui puisse être représentée sous forme d'une table. Prenons par exemple, lors d'une épidémie, le nombre de cas avérés testés chaque semaine, pour différents hôpitaux.

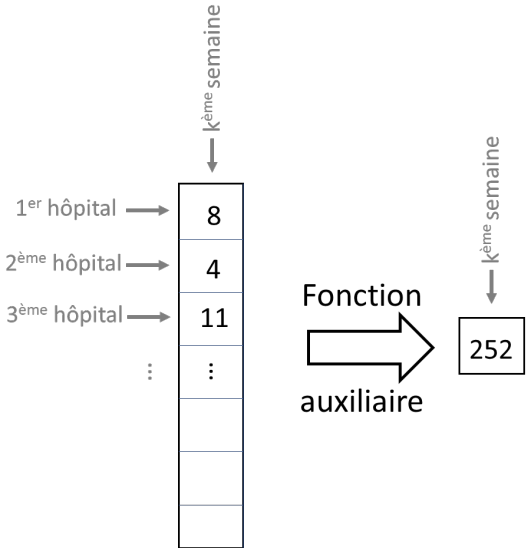


On cherche à écrire une **fonction principale** qui prenne en argument une table dont :

- Les lignes sont les cas avérés déclarés par différents hôpitaux (chaque hôpital correspond à une ligne)
- Les colonnes correspondent aux semaines.

Cette fonction principale est sensée permettre de tracer l'évolution de l'épidémie en fonction d'une zone qui regroupe différents hôpitaux. Elle doit renvoyer, pour chaque semaine donnée, le nombre de cas totaux sur la zone, c'est-à-dire la somme de tous les termes de la colonne.

On peut alors définir une **fonction auxiliaire**, qui prend en argument une colonne (c'est-à-dire une semaine) et calcule la somme des cas déclarés par chacun des hôpitaux pour cette semaine. La valeur renvoyée est alors un nombre (scalaire).



La fonction principale pourra alors s'appuyer sur cette fonction auxiliaire qui sera appelée plusieurs fois, avec à chaque fois une valeur d'argument différent.

Exemple :

```
Définissons f_auxiliaire(Colonne) :
    S ⟵ 0    # initialisation d'une somme toujours nulle
    Pour tout élément de la Colonne :
        S ⟵ S + élément
    Renvoyer S

Définissons f_principale(Table) :
    Sortie ⟵ Liste vide avec autant de colonnes que la Table
    Pour toute colonne_k de la Table :
        Remplacer la k^ème case de Sortie par f_auxiliaire(colonne_k)
    Renvoyer Sortie
```

La fonction auxiliaire sera appelée autant de fois qu'il y a de colonnes dans le tableau, c'est une sorte de factorisation d'un ensemble d'opérations.

Aux concours, les sujets sont intégralement pensés en termes de fonctions. Chaque question vous décrit précisément ce qu'on attend d'une fonction (quels sont ses arguments, qu'est-ce qu'elle renvoie). Pour la question *N*, vous pouvez supposer que toutes les fonctions écrites aux questions (1 à *N* – 1) sont correctes, même si vous n'êtes pas arrivés à les écrire, ou même si votre proposition est fausse. Ceci permet d'assurer que (presque) toutes les questions sont indépendantes, évaluées de manière distincte.

B / Écriture d'une fonction en *Python*

B. 1 / Déclaration d'une fonction : nom et arguments

En mathématiques, on ne définit pas une fonction par $f(x,y) = 3x + y$! Rigoureusement, il faut écrire :

$$f \begin{cases} \mathbb{R}^2 \to \mathbb{R} \\ x, y \to 3x + y \end{cases}$$

- La première ligne $\mathbb{R}^2 \to \mathbb{R}$ consistera à préciser le **type** du (ou des) **arguments** et de la (ou des) **sorties** de la fonction. En *Python*, on peut sous-entendre ces types, et c'est ce que nous ferons en CPGE.

- La seconde ligne $f : x, y \to 3x + y$ donne trois informations pertinentes : le **nom** de la fonction, le nombre d'**arguments**, et la **valeur** ce que **renvoie** la fonction.

En Python, on **déclare** une fonction de *n* arguments en écrivant :

```
def Nom_Fonction(Arg_1, Arg_2,…,Arg_n):      # 1ère ligne, dite « en-tête »
    Bla                                       # indentation (touche Tab) nécessaire
    Bla                                       # Dièze permet de faire des commentaires
    Bla
```

1 tab — corps de la fonction

B.2 / Retour et appel d'une fonction

Une fonction peut **renvoyer** une ou plusieurs **variables** avec la commande *return*.

Exemple : nom — argument(s)

```
1 def fonction_avec_un_nom_clair(x) :
2     return x**2 + 1
```

Appel de la fonction avec la valeur particulière *x* = 3

```
In [2]: fonction_avec_un_nom_clair(3)
Out[2]: 10
```

Ce qui est renvoyé

- Dès que la commande *return* est lue, on quitte instantanément la fonction dont c'est le retour.
- Une fonction peut également renvoyer plusieurs objets, en écrivant *return* objet_1, objet_2, …

Exemple :

```
>>> def g(v1, v2) :
        a = abs(v2 - v1)   # valeur absolue
        b = (v2 + v1)**2
        return a, b
```

Si on veut appeler la fonction pour le cas particulier `v1 ← 2, v2 ← 3`, et affecter les sorties à `s1` et `s2` :

```
>>> s1, s2 = g(2, 3)                    # affectation simultanée
```

- **Remarque :** Une fonction peut également ne rien renvoyer ! Ce serait par exemple le cas d'une fonction qui renomme des fichiers dans un dossier, ou modifie des fichiers externes, ou encore réalise un tracé (image, graphique). On parle alors plutôt de **fonction procédurale**. Nous nous en servirons, mais plus tard.

- Entre autres, si vous voulez quitter une fonction en plein milieu, sans qu'elle ne renvoie rien, vous pourrez taper

```
return None
```

C / Portée des variables, import de bibliothèque et structure imposée pour les scripts

Revenons à l'exemple de la page précédente. Rigoureusement, il convient de saisir la subtilité entre les deux notations :

- $f: x, y \to 3x + y$ signifie que pour tous x et y (réels ici), la fonction renvoie …
- $f(x, y) = 3x + y$ signifie que pour **un couple** (x, y) (de réels) fixé, la fonction appliquée à ce couple vaut …

J'insiste donc sur le fait que dans la 1ère notation, les arguments sont quelconques, dans la 2ème notation les arguments ont d'ores et déjà été fixés.

Plus spécifiquement en informatique, si après la définition de f on venait à affecter à x et y des valeurs singulières (par exemple :
```
>>> x = 3
>>> y = 2
```
… alors dans ce cas

- $f: x, y \to 3x + y$ reste une formule générale, ça reste la définition de la fonction
- Alors que $f(x, y) = 3x + y$ vaut 11, et est une constante … !

C. 1 / Variable locale

La plupart des fonctions dont on a besoin au quotidien ont déjà été écrites, optimisées, et sont partagées en ligne (*github.com* par exemple), ou en interne au sein de groupes de travail. Certaines de ces fonctions peuvent même ne pas être open source (le code peut être masqué). Normalement, une fonction bien écrite dispose d'une petite rubrique (sous forme de commentaire) qui précise le type des arguments attendus en entrée, et le type des sorties renvoyées. Ceci doit permettre à un utilisateur de se servir d'une fonction sans regarder le code du corps de fonction.

→ Il ne faudrait pas, alors, que les noms de variables utilisés dans la fonction entrent en conflit avec les noms de variables qu'a déclaré l'utilisateur (par exemple, qu'une variable interne dans la fonction écrase une variable utile à la personne qui appelle cette fonction).

→ La convention qui a été choisie est qu'une variable déclarée ou affectée à l'intérieur ont une portée limitée. Elle est « oubliée » dès la fin de l'exécution de la fonction.

On dit que la portée de ces variables est **locale.**

```
L1    def fonction(arg1, arg2, ... ) :  # les arguments sont des variables locales
L2          var_inter = ...      # déclarée dans la fonction : locale
L3          return var_inter
```

A chaque appel de cette fonction, on fait comme si on « remontait » dans le code à la ligne L1, et les variables `arg1, arg2, var_inter ...` prennent un jeu de valeurs particulières

Ces variables n'existeront qu'entre les lignes L1 et L3

Puis la fonction renvoie une **valeur** (pas une variable !) et détruit toutes les variables locales.

C. 2 / Variable globale

Il faut imaginer que dans un véritable projet informatique, vous allez avoir besoin de beaucoup de fonctions (auto-suffisantes ou qui s'appellent entre elles) et de bibliothèques de fonctions extérieures que vous importez. Ces fonctions seront généralement dans un fichier à part, alors que vous allez appeler les fonctions pour résoudre un problème particulier dans un fichier principal, qu'on appelle le main (anglicisme).

Les variables qui sont déclarées en dehors des fonctions sont dites **globales**.

Normalement, *Python* gère les conflits et protège les variables globales en empêchant qu'elles puissent être modifiées par le corps d'une fonction. En revanche, cela peut mener à des choses difficiles à comprendre.

Exemples : quelques casse-têtes, qu'il ne faudrait jamais écrire normalement …

Code 1

```
def fonc_test(x) :
    toto = 3*x
    return toto

print( fonc_test(2) )

print( toto )
print( x )

toto = 3
x = 4
print( fonc_test(5) )
print( toto, x )
```

→ affiche 6

→ erreurs, variables locales ☒ en dehors de la fonction

→ affiche 15

→ // 3,4

Code 2

```
def fonc_test2(x) :
    x = 3*x
    return x

x = 4
print( fonc_test2(5) )
print( x )
```

→ à l'avenir, ne jamais modifier un argument (sauf changement d'unité)

→ affiche 15

→ // 4

↑ Variables globales, non modifiées par la fonction, même si conflit de nom !

Code 3

```
K = 2 # portée globale

def f(x) :
    return K*x

K = 3 # portée globale
print(f(2))
```

→ 6

Code 4

```
def f(x) :
    K = 2 # locale
    return K*x

K = 3 # globale
print(K, f(2))
```

→ 3,4

Remarque : A l'intérieur d'une fonction, on peut « augmenter la portée » d'une variable qui serait normalement locale, avec la commande `global nom_de_la_variable` ce qui la passe en portée globale.

➔ L'usage « en autonomie » (sans que le sujet vous y force) de cette commande est **interdit**.

Code 5

```
def f(x) :
    global K
    K = 2 # globale
    return K*x

K = 3 # globale
print(f(2))
print(K)
```

→ 4

→ 2

Code 6

```
def f(x) :
    global sortie
    sortie = 3*x

f(3)

print(sortie)
```

→ RIEN !

→ 9

Pas de **return** (on affecte directement une variable externe, globale). Ces fonctions sont dites **procédurales**.

Code 7

```
def f(x) :
    global sortie
    sortie = 3*x

sortie = 'important' # on peut
  # écrire n'importe quoi ici
f(3)
print(sortie)

sortie = f(4) # erreur classique
print(sortie)
```

→ rien

→ 9, ancienne valeur perdue

→ None, f(4) ne renvoie rien, sortie détruite.

C. 3 / Structure de script imposée : pour les scripts de vos prochains TP et cette année en générale

```
### ESPACE D'APPEL DES BIBLIOTHÈQUES ###
# C'est ici que vous aurez vos « import … as … »

### ESPACE DE DÉCLARATION ET DÉFINITION DES FONCTIONS ###
# Ici, toutes les variables sont donc locales (sauf exception - expérimentés)

### SCRIPT PRINCIPAL ###
# À partir d'ici, les variables sont globales. C'est entre autres ici que :
# vous donnerez des valeurs numériques aux variables de votre problème
# vous tracerez des courbes, appellerez des fichiers externes
# vous appellerez les fonctions définies dans l'espace ci-dessus
```