Rappels : Fonctions

Chaddaï Fouché

2025

I] Motivation

Les **fonctions** en informatique sont essentiellement des morceaux de code, isolés du reste du programme, auxquels on donne un nom pour pouvoir les réutiliser à plusieurs reprises par la suite.

Il est théoriquement possible d'écrire des programmes sans utiliser jamais de fonctions, comme vous l'avez fait au début de votre apprentissage de Python (à part les fonctions pré-définies en Python comme `print`) et comme il était nécessaire de le faire aux débuts de l'informatique. Cela implique néanmoins de réécrire les mêmes suites d'instructions de façon répétée, à l'identique *sans erreur*, ce qui est à la fois un gâchis de travail et une source de bogues et de frustration lorsqu'on s'aperçoit qu'il y avait une erreur ou qu'on pouvait améliorer le code initial et qu'il faut corriger *sans erreur* chaque occurrence de cette opération dans son code...

Les Mathématiques avaient déjà depuis longtemps un mécanisme pour encoder une suite d'opérations répétables donnant toujours le même résultat sur une même valeur en entrée : les *fonctions mathématiques*. L'informatique a ainsi repris ce nom, y compris pour du code répétable dont le résultat n'est pas toujours le même (aléatoire, ou faisant des entrées sorties, etc) qui a parfois été nommé plus précisément **procédure** pour faire la distinction.

II] Notions et vocabulaire autour des fonctions

1) Code répétable

Initialement, une fonction est juste une suite d'instructions à laquelle on a donné un nom :

```
1  def dire_bonjour():
2      nom = input("Quel est votre nom ? ")
3      print(f"Bonjour {nom} !")
```

Ici on ***déf**init* une fonction `dire_bonjour` qui exécute une suite de deux instructions : un `input` dont le résultat est affecté à la variable `nom` et un `print` qui salue la personne nommée. Toutes les instructions indentées (toutes les lignes avec au moins autant d'espaces au début que la première ligne après le `:`) font partie de la fonction et *ne seront pas exécutées* immédiatement mais uniquement lors de l'**appel** de la fonction.

Bien qu'on utilise une syntaxe particulière pour définir une fonction, son nom est une variable comme une autre, donc obéissant aux mêmes règles (commence par une lettre ou _, continuant avec des lettres, chiffres ou _) et pouvant être utilisée ou affectée comme toute autre variable (par exemple on peut faire : `afficher = print` pour avoir un alias en français de la fonction `print`).

La particularité des variables qui contiennent une fonction est qu'on peut **appeler** la fonction contenue en mettant des parenthèses ouvrantes et fermantes après son nom :

```
print("avant l'appel")

# la ligne suivante appelle la fonction dire_bonjour
dire_bonjour()

print("après l'appel")
```

ce qui est équivalent à exécuter le code de la fonction à l'emplacement de son appel :

```
print("avant l'appel")

# la ligne suivante appelle la fonction dire_bonjour
nom = input("Quel est votre nom ? ")
print(f"Bonjour {nom} !")

print("après l'appel")
```

au détail près que la variable `nom` est locale à la fonction et n'affecte donc pas une éventuelle variable `nom` globale :

```
nom = "Noé"
# on appelle cette fonction et on entre "Paul" à l'invite
dire_bonjour() # ce qui affiche : Bonjour Paul !
print(nom) # cela affiche "Noé"
```

et n'est pas non plus disponible après l'appel de la fonction :

```
dire_bonjour()
print(nom) # plante avec l'erreur NameError: name 'nom' is not defined
```

2) Paramètres

Il est pratique de pouvoir répéter un code à l'identique mais il est encore plus courant de devoir répéter des instructions *presque* à l'identique mais avec quelques valeurs différentes à chaque fois.

Il serait possible d'utiliser des variables globales pour communiquer avec la fonction :

```
def dire_bonjour_à_nom():
    for i in range(1,4):
        print(f"{i}...", end=" ")
    print(f"Bonjour {nom} !")

nom = "Ado"
dire_bonjour_à_nom() # affichera : 1... 2... 3... Bonjour Ado !
```

Mais cela serait très peu pratique et générateur de bogues nombreux :

- il faudrait affecter les bonnes variables bien nommées préalablement à chaque appel, écrasant peut-être ainsi la valeur d'une variable globale ;
- cela interdirait d'imbriquer des appels de fonctions utilisant les mêmes variables globales simplement ;
- ce qui rendrait les fonctions récursives très difficiles d'usage ;

- etc.

Rien de ceci n'est insurmontable (on peut enregistrer la valeur de la variable globale temporairement dans une autre variable et la rétablir après l'appel de fonction) et c'est ainsi que procèdent les appels de fonctions en Assembleur (pour simplifier...).

Néanmoins dans nos langages de haut niveau on préfère utiliser un mécanisme bien plus élégant, lisible et pratique : les **paramètres** de fonction.

Il s'agit de variables locales à la fonction dont les valeurs peuvent être données à chaque appel de fonction. Pour ceci on met les noms des paramètres entre parenthèses séparés par des virgules après le nom de la fonction lors de la définition de la fonction :

```
def dire_bonjour_à(prénom, nom):
    print(f"Bonjour {prénom} {nom} !")
```

Et l'on met les valeurs que prendront ces paramètres dans les parenthèses, séparés par des virgules, lors des appels de la fonction :

```
dire_bonjour_à("Charles", "de Gaulle")
# affichera : Bonjour Charles de Gaulle !


dire_bonjour_à("Albert", "Einstein")
# affichera : Bonjour Albert Einstein !
```

ce qui est équivalent à :

```
prénom = "Charles"
nom = "de Gaulle"
print(f"Bonjour {prénom} {nom} !")
# affichera : Bonjour Charles de Gaulle !


prénom = "Albert"
nom = "Einstein"
print(f"Bonjour {prénom} {nom} !")
# affichera : Bonjour Albert Einstein !
```

toujours à ce détail près que `nom` et `prénom` sont des variables locales et donc n'existent, indépendamment de variables globales du même nom, que lors de l'exécution de la fonction.

Les valeurs **passées** à la fonction dans un appel sont les **arguments** de cet appel, à ne pas confondre avec les *paramètres* qui sont les noms donnés dans la définition de la fonction, par exemple ici :

- `nom` et `prénom` sont les paramètres de la fonction `dire_bonjour_à` ;
- `"Charles"` et `"de Gaulle"` sont les arguments de l'appel `dire_bonjour_à("Charles", "de Gaulle")` ;
- et `"Albert"` et `"Einstein"` sont les arguments de l'appel `dire_bonjour_à("Albert", "Einstein")`.

Notez également que les arguments sont les valeurs passées à la fonction, même si ces valeurs sont dans des variables :

```
firstname = "Michael"
lastname = "Jackson"
dire_bonjour_à(firstname, lastname)
```

Les arguments ici sont bien les valeurs `"Michael"` et `"Jackson"` même si l'on dit parfois abusivement qu'on a passé `firstname` et `lastname` à la fonction : ce qui est transmis à la fonction est bien la valeur, pas la variable.

3) Valeur de retour, valeur renvoyée

Lorsque l'on souhaite calculer une valeur à partir d'une autre (ou de plusieurs valeurs), de façon répétée, en utilisant toujours les mêmes formules ou algorithmes, on utilise une fonction avec une **valeur de retour** ou **valeur renvoyée**, qui *renvoie* une valeur calculée à partir des arguments envoyés, qui ramène une valeur après le *retour* depuis le code de la fonction dans le code appelant.

On peut considérer que la valeur de retour *remplace* l'appel à la fonction pour la suite des calculs, ainsi :

```
1  def ajoute_3(x):
2      return x + 3
3
4  print(ajoute_3(10))
```

est équivalent à :

```
1  print(13)
```

car la valeur de `x + 3` (ici 13 puisque `x` vaut 10 dans l'appel `ajoute_3(10)`) est calculée par la fonction puis le mot-clé `return` met fin à l'exécution de la fonction (même s'il reste du code après cette ligne dans la fonction !!) et remplace l'appel de la fonction par la valeur calculée.

Il est possible d'avoir plusieurs `return` dans une fonction : le premier exécuté met fin à la fonction et sa valeur sera celle renvoyée.

```
1  def statut_légal(age):
2      if age >= 18:
3          return "majeur"
4      else:
5          return "mineur"
6
7  print("Il est", statut_légal(20))       # affiche : Il est majeur
8  print("Celui-là est", statut_légal(10)) # affiche : Celui-là est mineur
```

Techniquement on ne peut renvoyer qu'une seule valeur depuis une fonction mais l'usage d'un type construit comme un p-uplet (`tuple`) permet aisément de contourner cette limitation en Python (et dans d'autres langages) :

```
1  def somme_et_produit(a, b):
2      return (a+b, a*b)
3
4  somme, produit = somme_et_produit(3, 5)
5  print(f"La somme de 3 et 5 est {somme} alors que leur produit est {produit}.")
6  # affiche : La somme de 3 et 5 est 8 alors que leur produit est 15.
```

Les fonctions en Python ont techniquement toujours une valeur de retour même si elle n'est pas indiquée : que ce soit par une absence de `return` ou par un `return` seul sans valeur lui succédant, la valeur spéciale `None` est renvoyée par défaut.

```
1  def pas_de_return():
2      print("Pas de return ici !")
```

```
3
4  print(pas_de_return())
5  # affiche :
6  # Pas de return ici !
7  # None
8
9  def juste_un_return():
10     while True:
11         return # met fin à la fonction donc à la boucle infinie
12
13 print(juste_un_return())
14 # affiche : None
```

III] Procédures et fonctions

1) Effets de bord

On appelle **effet de bord** d'une fonction toute interaction avec l'extérieur de la fonction autre que la valeur de retour. Par exemple la modification d'une variable globale ou de la valeur d'une variable passée en argument (pour les types mutables) ainsi que les entrées-sorties. Il s'agit d'une mauvaise traduction de l'anglais *side effect* qui devrait plutôt être traduit par *effet secondaire*, autrement dit un effet de la fonction autre que son effet principal de renvoyer une valeur.

Les effets de bord *incontrôlés* sont l'une des premières sources de bogues en programmation. Le *paradigme fonctionnelle* cherche donc à les minimiser ou même à les empêcher entièrement. Le *paradigme impératif* repose initialement sur l'exploitation des effets de bord mais pour éviter les nombreuses failles qu'ils peuvent provoquer, les bonnes pratiques modernes tendent à éviter les effets de bords dans les fonctions qui renvoient une valeur (en faisant des fonctions au sens mathématique) et à les limiter aux fonctions qui ne renvoient rien (les procédures).

2) Procédure

Ce nom est généralement réservé aux fonctions qui n'ont pas de valeur de retour (renvoie toujours `None` en Python). Elle n'ont donc d'effet que via leurs effets de bord.

On distingue particulièrement :

- les procédures qui modifient leurs arguments, par exemple les tris en place :

```
1  def tri_par_sélection(T):
2      for i in range(0, len(T) - 1):
3          # trouve l'indice du minimum dans T[i:]
4          indice_du_min = i
5          for j in range(i+1, len(T)):
6              if T[j] < T[indice_du_min]:
7                  indice_du_min = j
8
9          # place ce minimum à l'indice i, sa position finale
10         T[i], T[indice_du_min] = T[indice_du_min], T[i]
11
12 # l'effet d'une telle fonction ne peut être constaté que si la
13 # valeur passée en argument était dans une variable, de façon à
14 # y accéder après l'appel à la fonction.
15 tableau = [2,3,1,1,10,5]
```

```
tri_par_sélection(tableau) # renvoie None puisque pas de return
print(tableau) # affiche : [1,1,2,3,5,10]
```

- Les procédures qui modifient une variable globale. À noter que s'il est nécessaire d'affecter une nouvelle valeur à cette variable (et pas juste utiliser la variable ou modifier un type mutable dans une variable globale), il faut indiquer à Python qu'on souhaite affecter à la variable globale et pas créer une nouvelle variable locale en utilisant le mot clé **global** :

```
def met_au_pluriel():
    global objet
    # les chaînes de caractères (str) sont immuables,
    # on doit donc affecter à la variable leur nouvelle valeur
    # pour les "modifier"
    objet = objet + 's'
    # ou objet += 's' qui est aussi une affectation

objet = 'pomme'
met_au_pluriel()
print(f"Je mange des {objet}.") # affiche : Je mange des pommes.


# si l'on omet le global :
def change_nom():
    nom = "Jack"

nom = "Moriarty"
change_nom()
print(nom) # affiche : Moriarty
# car l'appel à change_nom a modifié la variable _locale_ nom


def ajoute_à_x(y):
    x = x + y

x = 5
ajoute_à_x(10) # plante avec :
# UnboundLocalError: cannot access local variable 'x' where
#     it is not associated with a value
```

Cette dernière fonction plante parce que l'affectation à `x` sans utiliser **global** signifie que Python considère `x` comme une variable locale à la fonction mais dans ce cas, elle n'a pas encore de valeur lorsqu'on cherche à calculer `x + y` car elle n'a rien à voir avec la variable globale `x` qui elle vaut 5.

- les procédures qui font des entrées-sorties (directement sur la console ou dans des fichiers, voire vers et depuis le réseau). Par exemple, le `dire_bonjour()` initial fait partie de cette famille.

3) Fonction (pure)

À l'opposé, on trouve les fonctions au sens mathématique, qui évitent tout effet de bord, parfois appelée fonctions pures pour les distinguer des "fonctions" historiquement utilisées en informatique, qu'elles aient des effets de bord ou non.

Un avantage certain de ce type de fonctions est leur prévisibilité : le résultat d'une fonction pure ne dépend que de ses arguments, pour les mêmes arguments elle renvoie toujours le même résultat.

Cela rend cette fonction beaucoup plus facile à tester : il n'est nul besoin de mettre en place un contexte, de se préoccuper d'intercepter les entrées et les sorties ou autre bricolage. Il suffit d'appeler la fonction avec certains arguments et de vérifier si la valeur de retour est bien celle attendue dans chaque cas.

Il est également plus facile de comprendre et de corriger une telle fonction, en effet son code peut être compris sans jamais penser au contexte (l'ensemble des variables globales) de l'application.

Quelques exemples :

```
# les fonctions mathématiques sont toujours pures
def fahrenheit_vers_celsius(fahrenheit):
    return (fahrenheit - 32) * 5 / 9

print(f"212°F font {fahrenheit_vers_celsius(212)}°C.")

# on peut également avoir des fonctions de tri qui
# produisent une version triée de leur argument plutôt
# que de le modifier, elles sont alors pures
def tri_rapide(L):
    if len(L) <= 1:
        return L.copy()
    else:
        pivot = L[0]
        plus_petits = [L[i] for i in range(1,len(L)) if L[i] < pivot]
        plus_grands = [L[i] for i in range(1,len(L)) if L[i] >= pivot]
        return tri_rapide(plus_petits) + [pivot] + tri_rapide(plus_grands)

# elles sont plus flexibles
print(tri_rapide([2,3,1,1,10,5]) # affiche : [1,1,2,3,5,10]
```

4) Compromis

Pour des raisons d'efficacité de l'exécution ou de simplicité des appels de fonctions, il est parfois préférable d'écrire une fonction qui tient un double rôle : qui a des effets de bord **et** qui renvoie une valeur. Ne vous interdisez pas cette possibilité dans vos projets, mais commencez par essayer de séparer les procédures qui interagissent avec le monde extérieur et les fonctions pures qui effectuent les calculs utiles à votre application. Le consensus est qu'une telle conception, sans s'astreindre à des contorsions contre-productives, est la plus facile à maintenir et améliorer ainsi que la moins susceptible d'avoir des bogues.

IV] Paramètres avancés (hors programme)

1) Argument(s) par défaut

Il est assez fréquent de définir une fonction dont l'un des paramètres aura toujours la même valeur, ou une valeur qui peut usuellement se déduire des autres arguments. Dans ce cas, Python permet de définir cette valeur "par défaut" et de l'omettre lors de l'appel à la fonction :

```
# on définit la fonction comme normalement mais le(s) argument(s) par
# défaut sont affectés au(x) paramètre(s) correspondant(s)
def saluer(prénom, nom = "Dupont"):
  print(f"Salut {prénom} {nom} !")

```

```python
# on peut utiliser cette fonction normalement en passant ses 2 arguments
saluer("Charles", "De Gaulle") # affiche : Salut Charles De Gaulle !

# si on ne passe qu'un seul argument, la valeur par défaut du deuxième
# est utilisée
saluer("Charles") # affiche : Salut Charles Dupont !

# parfois l'un des arguments peut avoir une valeur par défaut
# qui se déduit des autres arguments mais peut aussi être forcée
# à une valeur précise dans certain cas
def moyenne_et_comparaison(classe, comparés = None):
    """
    Calcule la moyenne de la classe, donnée sous forme d'un dictionnaire de
    paires (élève, note) et compare tous les élèves de la liste comparés
    à cette moyenne.
    Si comparés n'est pas fourni, tous les élèves sont comparés !
    """
    if comparés is None:
        comparés = sorted(classe.keys())

    moyenne = sum(classe.values()) / len(classe)
    for élève in comparés:
        note = classe[élève]
        if note < moyenne:
          print(f"{élève} est {moyenne - note} points en-dessous.")
        else:
          print(f"{élève} est {note - moyenne} points au-dessus.")

# on peut comparer quelques élèves par rapport à la moyenne
moyenne_et_comparaison(classe_nsi, ["Carl", "Leonhard"])

# ou juste voir toute la classe
moyenne_et_comparaison(classe_nsi)
```

⚠Attention à ne pas utiliser de valeurs mutables comme argument par défaut (le plus classique étant la liste vide `[]`) car cette même valeur va être réutilisée, y compris après avoir été modifiée...

```python
# vous voulez définir une fonction qui ajoute
# un élément à la fin d'une liste et la renvoie
# ou crée une liste à un élément sinon :
def ajout(élément, liste = []):
  liste.append(élément)
  return liste

print(ajout(3)) # affiche [3]
print(ajout(10)) # affiche [3, 10]
```

Comme vous le voyez, la liste vide par défaut n'est pas réinitialisée à chaque appel de fonction, la même liste créée lors de la définition de la fonction est réutilisée et accumule les éléments.

Il est très rare qu'il soit correct d'utiliser une valeur mutable comme argument par défaut. La bonne solution consiste généralement à utiliser `None` comme argument par défaut et dans le corps de la fonction, initialiser le paramètre à la valeur correcte s'il vaut `None`, par exemple :

```python
def ajout(élément, liste = None):
  if liste is None:
      liste = []
  liste.append(élément)
  return liste
```

2) Argument nommé

Parfois il y a trop de paramètres, certains avec des arguments par défaut, et ils ont souvent le même type, dans ce cas il devient très difficile de comprendre un appel à la fonction.

```python
# par exemple les formules physiques :
def conductivité(perméabilité, masse_volumique, viscosité, g = 9.8):
    return (perméabilité * masse_volumique * g) / viscosité

conductivité(2.3, 1.5, 0.7)
```

Dans cet appel, il est très difficile de savoir à quoi correspond chaque valeur, apprendre par cœur l'ordre des paramètres n'est pas optimal. Il est préférable d'indiquer le nom de chaque argument lors de l'appel :

```python
conductivité(perméabilité=2.3, viscosité=0.7, masse_volumique=1.5)
```

On peut même alors se permettre de ne pas respecter l'ordre de la définition comme ici !

Vous avez déjà utilisé ceci avec la fonction `print` dont les paramètres `end` et `sep` ont des arguments par défaut (`"\n"` et `" "` respectivement) et ne sont utilisés (et utilisables) que comme des arguments nommés.