

TP6 : Les fonctions

Notion d'objet mutable et objet non-mutable

En Python, tout est objet et il existe deux types d'objets, les **mutables** (listes, dictionnaires, ...) que l'on peut modifier et les **non mutables** (strings, int, floats, tuples, etc) qui demeurent constants et ne peuvent être changé.

Python étant un langage à **typage dynamique**, les variables sont créées directement lors de leur initialisation et typées en fonction de la valeur qui leur est affectée. De la même manière pour les variables numériques, des objets numériques adaptés sont créés en mémoire lors de leur initialisation. Les fonctions intégrées «**type**», pour connaître le type d'un objet, et «**id**» pour l'identifier de manière unique, permettent de comprendre comment Python prend en charge les types numériques.

```
>>> x=3
>>> print("valeur de x : %d" % x)
Valeur de x : 3
>>> print("identifiant de x : %d" % id(x))
identifiant de x : 1362216688
>>> x+=4
>>> print("valeur de x : %d" % x)
Valeur de x : 7
>>> print("identifiant de x : %d" % id(x))
identifiant de x : 1362216816
```

En exécutant le code ci-dessus, on constate que la valeur de «x» a bien changer pour la valeur 7. On constate que l'identifiant de l'objet mémoire pointé par «x» a également changé. Il ne s'agit donc plus du même objet mémoire.

En python, les opérations arithmétiques génèrent des résultats stockés dans des **objets mémoires uniques**, chaque valeur arithmétique étant donc unique. Lors de l'affectation, l'étiquette «x» libère l'objet numérique initial pour pointer sur l'objet représentant le résultat. Les objets de type numérique ne peuvent donc être modifiés, ils sont non-mutables. Il en est de même pour les chaînes de caractères dans lesquelles chaque caractère est unique.

Exercice 1 :

- Initialisez une liste ne comprenant que trois zéros à l'aide de la commande Python « `L1 = [0]*3` » qui concatène trois listes à élément unique « `[0]` » en une seule et affichez la liste obtenue ainsi que son type et son id.
- Pour chaque élément de la liste précédente, affichez son type et son identifiant à l'aide des fonctions « `type()` » et « `id()` ». Qu'est-ce que vous remarquez ?
- Modifiez l'élément en deuxième position pour l'augmenter de 1 avec l'expression « `L1[1] += 1` » puis affichez de nouveau la liste ainsi que son type et son id. Est-ce que l'id de la liste à changer ?
- Revérifier les identifiants de chaque élément de la liste, est-ce que c'est un nouvel objet qui a été créé suite à l'incrémentation ou bien c'est toujours le même ? que pouvez-vous en conclure sur les éléments de la liste

Rappels de cours – Définir une fonction en Python

La complexité de la tâche à accomplir par un programme peut conduire à la décomposer en **sous-tâches** s'occupant chacune d'un traitement spécifique et à les coordonner ensemble. Ce découpage fonctionnel améliore la lecture du programme et facilite sa maintenance. Cela permet également de réutiliser certains traitements sans avoir à les réécrire.

Les fonctions intégrées du langage, les fonctions des modules importés, les méthodes des objets manipulés sont autant de **sous-traitements** de ce type. En les associant, on compose le programme.

Pour créer une fonction, il suffit de créer un bloc de code en utilisant le mot réservé « `def` » et lui donner un nom. On indique entre parenthèses, les **arguments**, séparés par des virgules, utiles aux traitements à effectuer par ce sous-programme. Bien évidemment, on peut définir des fonctions sans argument.

La signature d'une fonction permet au langage de l'identifier. En python, la signature d'une fonction se résume à son nom. Par conséquent, deux fonctions différentes ne peuvent porter le même nom. Si vous donnez à une fonction le nom d'une autre, vous écrasez l'ancienne définition de la fonction par la nouvelle.

```
def NomDeLaFonction (arg1, arg2, ...):  
    instruction 1  
    instruction 2  
    return valeur
```

Le mot réservé « `return` » permet à la fonction le renvoi d'une valeur au programme appelant. Il est facultatif et n'est utilisé que dans le cas où le renvoi d'une valeur est attendu. Pour utiliser la fonction, il suffit de l'invoquer en lui fournissant les valeurs à affecter à chaque argument, soit directement, soit par l'intermédiaire de variables préalablement définies et initialisées.

🚦 Remarque : Python transmet les arguments à la fonction par **référence**. Cela signifie que ce n'est pas simplement la valeur de l'argument qui est transmis mais l'objet mémoire lui-même. Par conséquent, si la fonction manipule l'argument en le modifiant, l'objet initial sera effectivement modifié à la fin de l'exécution de la fonction et aura une nouvelle valeur pour toute la suite du programme. On parle d'effet de bord de la fonction.

Exercice 2 :

Créez une fonction nommée « `ajouter_elt(lst, elt)` » qui prend en argument une liste et un élément et qui retourne la liste « `lst` » agrandie de l'élément « `elt` » conformément au code ci-après.

```
# déclaration de la fonction ajouter_elt(liste, elt):  
def ajouter_elt(lst, elt):  
    lst.append(elt)  
    return lst
```

- Créer la liste « `lst1` » initialisée avec la valeur « `[0, 1, 2]` ».
- Créez une seconde liste « `lst2` » initialisée par la fonction « `ajouter_elt` » avec comme argument la liste `lst1` et sa longueur.
- Pour chacune des listes « `lst1` » et « `lst2` », affichez la liste, son type et son identifiant. Qu'est-ce que vous remarquez ?

Exercice 3 : Valeur par défaut et référence

Python offre la possibilité de définir une valeur par défaut aux arguments d'une fonction s'ils sont omis lors de son utilisation. Cela permet d'initialiser une référence pour toute la durée du programme. Par exemple :

```
def mafct(a, b=5):  
    return a+b  
  
x=3  
print(mafct(x, 3)) # affiche 6  
print(mafct(x)) # affiche 8 car b n'est pas transmis, donc la valeur  
par défaut est utilisée
```

Pour une fonction à plusieurs arguments, si vous prévoyez une valeur par défaut pour un argument, vous devez prévoir une valeur par défaut pour chacun des arguments qui le suivent ! Ainsi les arguments avec valeur par défaut doivent être à la fin de la liste des arguments.

Supposons que l'on adapte la fonction de l'exercice 2 en définissant des valeurs par défaut aux arguments de la fonction comme suit :

```
# déclaration de la fonction ajouter_elt(lst=[0, 1, 2], elt=3)
def ajouter_elt(lst=[0,1,2], elt=3):
    lst.append(elt)
    return lst
```

- a) Quel serait le résultat de l'instruction `print(ajouter_elt())` ?
- b) Répéter l'instruction précédente, qu'est-ce que vous remarquez ? Expliquez le résultat obtenu en vous aidant de la fonction `id()`
- c) Déclarer maintenant la fonction `ajouter_carac` qui prend en paramètres les deux arguments `ch` et `elt` ayant comme valeur par défaut `"abc"` et `"d"` respectivement, puis renvoie la concaténation de la chaîne `ch` avec l'élément `elt`
- d) Quel serait le résultat de l'instruction `print(ajouter_carac())` ?
- e) Répéter l'instruction précédente, qu'est-ce que vous remarquez ? Expliquez le résultat obtenu en vous aidant de la fonction `id()`

Exercice 4 :

On considère le programme python suivant:

```
import random

## Fonction generer(nbr, vmin, vmax) pour générer un tableau de 'nbr'
valeurs comprises entre 'vmin' et 'vmax'

...

## Fonction combienInferieur(table, vseuil) pour compter le nombre de
valeurs d'un tableau 'table' inférieures à la valeur 'vseuil'

...

nb = 100

print(f"Générer {nb} nombres entiers entre 0 et 100")
tab = generer(nb, 0, 100)
tab.sort()
print(tab)
total = combienInferieur(tab, 25)
print(f"Il y en a {total} inférieurs à 25")
```

- a) Pour que ce programme fonctionne, il manque les fonctions « `generer()` » et « `combienInferieur()` ». Proposez le code de ces fonctions pour que ce programme fonctionne.
- b) Vous allez à présent rendre votre programme interactif en donnant à l'utilisateur le choix de préciser le nombre de valeur à générer, l'intervalle (vmin et vmax) ainsi que le seuil. Pour le seuil, vous poser la question suivante à l'utilisateur « vous voulez préciser le seuil ? ». S'il répond 'Oui' ou 'O' alors il faut prendre le seuil qui sera précisé. Si la réponse est 'Non' ou 'N', la valeur 30 sera appliquée par défaut comme seuil.

Exercice 5 : Palindrome - version avec les fonctions

L'exercice sur les palindromes est vu dans le TP précédent (exercice 3). Cette fois-ci vous allez permettre à l'utilisateur de rentrer des mots/phrases avec des caractères accentués ou tout caractère spécial et indiquer s'il s'agit d'un palindrome ou pas. Le programme devra en particulier tenir compte de la présence éventuelle de majuscules, d'accents, d'espaces, de caractères typographiques et des signes de ponctuations. Vous devez donc prévoir une fonction pour nettoyer le texte des caractères spéciaux, espaces et autres ponctuations. Prévoir également une fonction pour remplacer les caractères accentués par leur caractère de base (ex : « éèëë » donne « e »). Prévoir enfin une fonction pour vérifier si c'est un palindrome ou pas.

Point d'étape 1 à faire valider (exercice 4 et 5)