Chapitre 4 : fonctions et les variables

Table des matières

1 Manipulations de base sur les fonctions :

Ce qui suit reprend ce qui a été fait au T.P. 3.

1.1 La définition d'une fonction

La syntaxe de définition de fonctions en PYTHON est très simple. Si on veut définir une fonction mathématique, par exemple $f : x \mapsto x^2 + 2x + 1$, on rentrera, dans la zone d'écriture de fichier de I.E.P. :

```
def f(x):   # bien noter les : obligatoires et l'indentation ligne suivante
    return x**2+2*x+1
```

De même pour les années bissextiles

```
def bissextile(annee):
    test=((annee%4==0) and ((annee%100!=0) or (annee%400==0))
    return test
```

Notez bien que *toutes* les instructions servant à la définition de la fonction sont indentées.

Ce dernier code est la *définition d'une fonction* dont le nom est `bissextile` qui prend un argument appelé `annee`, et qui *retourne* ou *renvoie* une valeur appelée `test` à l'intérieur de la fonction.

1.2 L'utilisation d'une fonction

Que faire des scripts de définition données au § 1.1 ? On les *exécute*, et une fois ce script exécuté, les fonctions sont en mémoire et on peut *appeler* chaque fonction, autant de fois qu'on veut *dans le shell ou dans un autre morceau de programme* pour avoir une réponse. Dans le shell, comme suit :

```
>>> f(2)
9
>>> bissextile(2000)
True
>>>bissextile(2014)
False
```

Bien retenir ces trois temps : définition de la fonction, exécution du script, appels de la fonction.

1.3 Comment on peut définir une fonction avec plusieurs arguments

Une fonction peut avoir plusieurs arguments (variables en entrées) et plusieurs sorties. Pour les fonctions de deux variables, par exemple, on connaît déjà la construction de *tuple*

```
def f(x1,x2):
    return x1*x2,x1+x2
```

La fonction suivante agit sur une chaîne de caractères.

```
def tiret(mot_1,mot_2):
    "met un tiret entre deux chaînes de caractères"
   return mot_1+"-"+mot_2
```

La seconde ligne sert de *documentation* et s'obtient avec la commande `help`.

```
>>> help(tiret)
Help on function tiret in module __main__:

tiret(mot_1, mot_2)
    met un tiret entre deux chaînes de caractères
```

1.4 L'importance du `return`

a) La fonction suivante affiche bien un résultat, sans `return` :

```
def tiret2(mot_1,mot_2):
    print(mot_1+"-"+mot_2)
```

Quelle différence avec `tiret` ? C'est qu'on ne peut pas stocker, utiliser, le résultat de `tiret2`. Cette fonction `tiret2` ne fait qu'un *affichage*. Ainsi comparer :

```
parole=tiret('bla','bla')
print(parole)
bug=tiret2('bla','bla')
print(bug)
```

Le `return` permet que votre fonction retourne vraiment une valeur, que vous pourrez stocker.

b) L'execution de la commande `return` provoque la *sortie* de la fonction, et par exemple d'une boucle.

On a déjà vu l'algorithme suivant au chap. 3, ici on le met dans une fonction.

```
def f(N):
    n=0
    while 2**n <=N:
        n=n+1
    return n
```

Voici une autre façon de faire la même chose :

```
def f_bis(N):
    n=-1
    while 1==1: # condition toujours vraie ! On peut mettre, mieux, while True
        n=n+1
        if 2**n>N:
                return n  # fait sortir de la fonction donc de la boucle.
```

2 Plus de précision sur les arguments d'une fonction

Définition : Les arguments d'une fonction sont les variables qui doivent/peuvent être mises en entrées pour utiliser la fonction. On a déjà vu des fonctions avec un ou plusieurs arguments.

2.1 Une fonction peut très bien ne pas avoir d'arguments

Comme celle-ci

```
def sois_poli():
    print('bonjour monsieur')
```

Noter que cette fonction *ne retourne pas non plus de valeur* mais fais seulement un affichage `print`. Ainsi

```
>>>a=sois_poli()
# ne donne pas de message d'erreur mais
>>> print(a)
None
```

Presque la même mais qui retourne une valeur :

```
def je_suis_galant():
    print('bonjour mademoiselle')
    return 'un bouquet de fleur'
```

Dans ce cas, on obtient comme affichage lors de l'appel de cette fonction dans le shell :

```
>>>>> je_suis_galant()
bonjour mademoiselle
'un bouquet de fleur'
```

Mais comme valeur de retour, on obtient :

```
>>>a=je_suis_galant()
>>>print(a)
un bouquet de fleur
```

Noter une fois encore la différence entre *l'affichage* provoqué par l'appel de la fonction dans le shell d'un côté, et *la valeur de retour* de la fonction de l'autre.

2.2 Une fonction peut avoir un (des) arguments optionnels

Considérons l'exemple suivant :

```
def Carl_Friedrich(N=101):
    S=0
    for i in range(N):
        S=S+i
    return S
```

Ici, l'affectation `N=101` à l'intérieur de la liste des arguments de `Carl_Friedrich` dit que :

— Si on appelle cette fonction sans préciser de valeur d'argument, elle fera le calcul pour `N=101`, ainsi :

```
>>> Carl_Friedrich()
5050
```

— Si on appelle cette fonction en précisant une valeur d'argument, on modifiera la valeur du `N` dans l'algorithme, ainsi :

```
>>> Carl_Friedrich(10)
45
```

3 Les variables à l'intérieur de la déf. d'une fonction :

3.1 La notion de variable locale

a) **Un premier exemple pour sentir ce qu'est une *variable locale* :**

```
def fonction_bete():
    a=1
    print('ai-je mis la valeur',a,'quelque part ?')
```

Si on fait `print(a)` dans le shell, la réponse sera :
L'affectation `a=1` n'existe pas dans la mémoire du `shell`. Pourtant elle a existé pendant le déroulement de la fonction, puisque sinon l'instruction `print` n'aurait pas donné le résultat obtenu. C'est en ce sens qu'on dit que la variable `a` *affectée pendant la définition de la fonction* est une *variable locale* à la fonction.

Les variables *créées dans le code de déf. d'une fonction* sont dites *locales*. Elles occupent un espace mémoire séparé appelé *espace local*.

b) **Parallélisme avec les variables muettes des maths :** dans la fonction `Carl_Friedrich` définies ci-dessus, la variable `i` est locale : c'est bien l'analogue de la variable de sommation en maths.
On va voir, avec l'exemple suivant, que la variable `S` est aussi locale, et cela c'est plus surprenant.

c) **Un troisième exemple pour aller un peu plus loin :**

```
def renvoie_un():
    a=1
    return a
```

Là on pourrait se dire, ah cette fois, elle renvoie `a`, donc si on exécuter `renvoie_un()` on aura `a=1`. Il n'en est rien !
La fonction *retourne la valeur 1* c'est tout, et la commande `a=1` n'a pas été opérée dans la mémoire principale. Elle a eu lieu seulement dans *l'espace local* de la fonction, qui s'est vidé à la fin de l'exécution.

C'est le *piège de la variable de retour* : la *valeur* de la variable de retour est bien communiquée à l'espace mémoire extérieur à la fonction (espace global) mais pas le *nom* de cette variable !

3.2 La bonne cohabitation des variables locales et globales

a) **Un code avec deux `a` qui cohabitent très bien :**

```
a=1
def pauvre_fonction():
    a=2
    return a
print(pauvre_fonction())
print(a)
```

Résultat ?

Moralité : Dans *l'espace local* de la fonction, on peut utiliser une variable locale qui s'appelle `a` en lui affectant la valeur `2` même s'il existe déjà une variable globale `a` avec la valeur `1` dans l'espace global. Ces deux affectations n'interfèrent pas :

- A l'intérieur de la déf. de la fonction, ici seule l'affectation locale `a=2` compte
- A l'extérieur de la déf. de la fonction, seule l'affectation globale `a=1` compte.

b) Quand il n'y a pas de redéfinition en local de la variable, la fonction utilise la variable définie en global. Comme dans l'exemple suivant :

```
mon_pi=3.14
def perimetre(R):
    return 2*mon_pi*R
```

Moralité : Lorsque l'interpréteur Python doit évaluer le contenu d'une variable, d'un nom, il le fait suivant **l'ordre de priorité suivant** : **espace local puis espace global**.

- d'abord l'espace local : il regarde si la variable a été définie localement (cf. exemple du a)).
- si la variable n'a pas été définie localement, il regarde si la variable est définie dans l'espace global comme dans `perimetre`.

Remarque (pas essentielle ici) On peut compléter cette liste de priorité en disant que : l'espace local est prioritaire sur l'espace global qui est lui-même prioritaire sur l'espace interne.

Ce dernier espace interne est celui qui contient des noms de variables ou de fonctions déjà définies par PYTHON et qu'on peut redéfinir.

3.3 Comment modifier une variable globale dans la déf. d'une fonction ?

On vient de voir au 3.2 b) qu'on peut lire le contenu d'une variable globale dans une fonction. Mais comment modifier ce contenu pour que cette modification apparaisse dans l'espace global ?

Avec la spécification : `global`, comme dans l'exemple suivant :

```
pi=3.14
def ameliore_la_trigo():
    global pi
    pi=3.1415
```

L'exécution de cette fonction changera la variable globale `pi`.

Pour une bonne pratique de la programmation : les variables globales devraient être réservées à des *constantes* du programme, qu'on n'a pas besoin de modifier. Il est préférable de leur donner un nom long ou en tous cas explicite. Suivant cette recommandation, la spécification `global` de PYTHON ne sera pas utilisée souvent !

3.4 Cas d'un argument d'une fonction

Par simplicité dans ce qui suit, on ne prend qu'un argument, mais le raisonnement vaut aussi bien pour chacun des arguments d'une fonction ayant plusieurs arguments.

3.4.1 L'argument d'une fonction est une variable LOCALE

Lorsqu'on définit une fonction, le nom de(s) variable(s) dans l'argument est *LOCAL*

```
def ajoute(x,y):
    return x+y
```

Les noms de variables `x` et `y` qui apparaissent ici peuvent être utilisés ailleurs sans problème, par exemple :

```
x=2
y=3
ajoute(5,6)
print(x)
print(y)
```

3.4.2 On peut utiliser et réaffecter en local l'argument de la fonction

```
def diminue(a):
    a=a-1
    return a
```

Cette fonction n'utilise qu'une seule variable, son argument, qui a pour nom `a` en local.

3.4.3 Mais une fonction NE PEUT PAS réaffecter son argument en global

Que vaut `a` à l'issue du code suivant :

```
a=6
diminue(a)
```

Moralité : si on veut vraiment modifier la variable `a` avec la fonction `diminue`, il faut réaffecter dans `a` le résultat de `diminue(a)` autrement dit exécuter :
`a=diminue(a)`
Avertissement : Ce qui précède doit être nuancé : si la variable `a` est une liste, ou plus généralement un objet *mutable* (modifiable), la situation est plus subtile, nous y reviendrons. Le titre de ce paragraphe est donc valable pour les arguments de type `int, bool, float, str,tuple` mais pas tout à fait pour les listes.

Culturel pour nous à ce stade : Le fonctionnement précédent s'appelle le *passage par valeur*. Dans le code ci-dessus, on avait une variable globale `a` avec comme valeur `6`. Lorsqu'on exécute `diminue(a)` le programme crée un espace mémoire local à la fonction avec une variable locale que j'appellerai pour simplifier `a.locale` (placée à un autre endroit que la variable `a` précédente) et il *passe* la *valeur* de l'entrée `a` à la variable locale `a.locale`. Après la variable globale `a` n'intervient plus, en particulier elle ne peut pas être modifiée !

Pour comprendre l'avertissement : Il faudra déjà en dire plus sur ces objets mutables et non mutables. Mais on connaît déjà des fonctions, un peu étranges, qui opère sur les listes en les modifiant.

Par exemple si `L=[15,3,5]` la commande `del(L[1])` qui est une fonction il est vrai à la syntaxe un peu bizarre, modifie la liste `L`.

4 Valeurs et références : davantage sur l'affectation des variables

4.1 L'identifiant mémoire ou référence

Si on rentre `a=3`, la variable `a` contient l'entier 3. La commande `id(a)` donne *l'identifiant mémoire* où est stocké `a`. Ce sera un nombre assez long.... essayez... Cela nous renvoie au cours du chapitre 1 : chaque mémoire a un *numéro d'emplacement* (son `id`) et un contenu (ici `3`)

En fait, en PYTHON, même si on ne stocke pas un nombre dans une variable, si un nombre intervient, il lui donne une adresse mémoire. Ainsi essayez :

```
id(3) # 3 n'est pas stocké dans une variable, mais dès qu'on l'introduit il aura
#une adresse.
a=3
id(a)
b=3
id(b)
```

Ici, les noms de variables `a` et `b` pointent tous vers la même adresse mémoire qui est celle où est stockée le nombre 3.

4.2 Un exercice sur l'affectation et la copie pour les types simples

Exercice 1. a) Que pensez-vous que donne le programme suivant :

```
a=3
b=a
a=2
print(b)
```

Vérifions !

b) Pourquoi ce qui suit est-il révélateur de ce que fait l'affectation ?

```
a=3
id(a)
b=a
id(a)
id(b)
a=2
id(a)
id(b)
```

Commentaires avec des dessins :

4.3 Un exercice fondamental sur l'échange de deux variables

Exercice 2. On a deux variables `x` et `y` avec un certain contenu.

a) Pourquoi le code suivant ne convient-il pas ?

```
x=2 # pour fixer les idées
y=3 # sur le contenu des variables
x=y, # les deux variables contiennent la valeur 3
y=x
```

Dans ce qui suit on propose trois méthode assez différentes pour y arriver :

b)**(M1)** *La méthode standard en informatique*, qui marche dans *tous les langages de programmation* est d'utiliser une troisième variable, auxiliaire. Comment faire ?

c) **(M2)** *Une méthode plus proche d'opérations que nous ferons en mathématiques* se résume dans la suite d'instruction suivante :

```
x=x+y,
y=x-y,
x=x-y
```

(i) Commentez cette suite d'instruction, qu'obtient-on à la fin.

(ii) Appliquez-la à `x=1/3` et `y=1/2`. Que pensez-vous du résultat ?

d) **(M3)** *Une méthode propre à* PYTHON *: l'affectation multiple*

A l'aide de l'affectation en couple vue au paragraphe sur les tuples, comment faire ?

4.4 Le cas différent de la copie des listes

Exercice 3. a) Reprenons le schéma de l'exercice 1 mais pour deux listes. Autrement dit, considérons le code

```
a=[1,2]
b=a
a[0]=3 # ceci est une modification pas une réaffectation de a
print(a)
print(b) # b est modifié comme a
```

Quelle différence avec l'exercice cité ?

b) Quelle raison trouver au comportement du a) ? Reprendre la question de l'exercice 1 b) avec le cadre du a).

D'une manière générale, en informatique, on dit que les objets qui se comportent comme les listes ici sont *mutables* ou *muables*. Ils peuvent changer de valeur sans changer d'identifiant.

A l'intérieur d'un objet liste, chaque objet a son identifiant... ainsi `id(a[0])` lui changera dans l'affectation précédente....

Moralité : en faisant `b=a`, on crée un *alias* de la liste `a` qui restera toujours comme `a` quelque soient les modifications de `a`.

Comment créer une copie un peu plus autonome ? A l'aide des commandes d'extractions dans une liste (slicing).

Exercice 4. a) Entrez le code suivant :

```
a=[10,6,7]
id(a)
b=a[0:3]
print(b)
id(b)
```

Que peut-on en déduire sur `b` si on modifie `a` ? Le vérifier.

b) **Remarque :** Plutôt que `a[0:3]`, on peut taper `a[:]`.