ITC – cours n°5

Fonctions

I. Définition et utilisation d'une fonction

1.1. Principe et syntaxe

Une fonction en informatique est un moyen de grouper en un « bloc » une séquence d'instructions, cette séquence pouvant être ajustée à travers un ou plusieurs paramètres qu'on lui fournit (c'est d'ailleurs ce qui fait l'intérêt des fonctions). Puisque cette séquence d'instructions sera appelée à plusieurs reprises, on lui donne un nom (une étiquette) et elle sera stockée quelque part en mémoire. La syntaxe de définition usuelle est la suivante :

```
def nom_de_la_fonction(arguments):
    instruction_1
    ...
    instruction_n
    return résultat  # ligne optionnelle - voir plus loin
```

Comme pour les branchements conditionnels ou les boucles, on voit apparaître un **bloc** d'instructions, ouvert par les deux-points et l'indentation, et dont la fin est annoncée par la fin de l'indentation.

Pour appeler la fonction, on utilise son nom suivi de l'opération d'appel de fonction[1] `()` (les parenthèses), et on donne les paramètres entre les parenthèses. Nous l'avons déjà croisé, par exemple la fonction `len` donne la longueur d'une liste `x` suite à l'instruction `len(x)` .

Il ne faut pas confondre **la fonction elle-même** (dans notre exemple `len`) et le **résultat de son appel** (dans notre exemple `len(x)`). La différence visible est la présence ou l'absence d'opérateur d'appel de fonction, et on peut bien voir dans un terminal qu'il s'agit de deux objets différents :

```
In [1]: x = [2, 3, 5]
In [2]: len(x)
Out[1]: 3
In [3]: len
Out[2]: <function len(obj, /)>
```

[1] Le nom « opérateur d'appel de fonction » n'est pas officiel, mais il est pratique pour le penser.

Nous avons rencontré quelques fonctions dont la syntaxe est légèrement différente, typiquement `append` et `pop` pour les listes. Nous ne rentrerons pas dans les détails de cette forme, mais on peut retenir que `maliste.append(truc)` est équivalent à `list.append(maliste, truc)` , donc que l'objet présent avant le point est équivalent à un premier paramètre pour une fonction appelée en réalité `list.append` (et de même pour les autres). Cette subtilité n'est pas à retenir.

Le PEP8 recommande la mise en forme suivante :

- ☞ pas d'espace contre les parenthèses ;
- ☞ une espace après les virgules dans les paramètres ;
- ☞ **deux** lignes blanches avant et après la définition d'une fonction ;
- ☞ on peut sauter une ligne de temps en temps dans le corps d'une fonction, pour séparer des blocs cohérents entre eux.

1.2. Fonctions et procédures

Revenons sur le cas de la fonction `len` : l'appel de cette fonction exécute des instructions[2] telles que l'expression `len(x)` corresponde à la longueur du conteneur. On peut donc considérer que cette fonction prend en paramètre le conteneur et **calcule** sa longueur : cette longueur est donc **le résultat** de l'appel à la fonction. Pour garder ce résultat en mémoire au-delà de l'appel, on doit la récupérer dans une variable, par exemple `n = len(x)` .

Le fait qu'une fonction renvoie un résultat est associé au mot-clé `return` : l'expression après ce mot-clé sera évaluée et c'est celle qui sera renvoyée. Par exemple, pour une fonction qui fait la somme des éléments d'une liste d'entiers fournie :

```
def somme_liste(li):
    total = 0       # li est définie avant cette ligne, nous y revenons
    for elt in li:  #    au prochain paragraphe
        total += elt
    return total
```

Notons que la liste fournie **n'est pas modifiée** lors de l'appel à cette fonction :

```
In [1]: x = [1, 4, 6]
In [2]: somme_liste(x)
Out[1]: 11
In [3]: x
Out[2]: [1, 4, 6]
```

Cette fonction ne fait que calculer un résultat et le renvoyer, sans modifier les variables existantes avant son appel : on dit qu'elle ne génère pas **d'effet de bord**.

[2] Non connues de nous, mais peu importe : nous sommes exclusivement **utilisateurs** de la fonction `len` , seul son comportement nous intéresse

Pour voir la différence, prenons le cas d'une fonction qui a pour intérêt principal un effet de bord :

```
def remise_à_zéro(liste):
    n = len(liste)  # également une subtilité ici, on y revient tout de suite
    for i in range(n):
        liste[i] = 0
    # optionnel : return None
```

Cette fonction **modifie la liste donnée en entrée** : par exemple,

```
In [1]: x = [3, 2, 1]
In [2]: x
Out[1]: [3, 2, 1]
In [3]: remise_à_zéro(x)
In [4]: x
Out[2]: [0, 0, 0]
```

Une telle fonction ne calcule et ne renvoie rien : on note d'ailleurs l'absence du mot-clé `return` dans sa définition. Pour souligner cette caractéristique, une telle fonction est appelée **procédure**.

- ☞ L'absence de `return` est équivalent à `return None` : Python rajoutera toujours implicitement le `return None` si on ne met pas de `return`.
- ☞ Certaines fonctions renvoient une valeur et on un effet de bord en plus : nous avons déjà rencontré pop qui modifie une liste en enlevant le dernier élément et le renvoie. Cependant, on peut considérer en général qu'il s'agit d'une mauvaise pratique : il vaut mieux savoir en lisant un code si on appelle une fonction qui calcule quelque chose (sans effet de bord), ou si on appelle une procédure qui produit un effet de bord.

Notons enfin qu'on peut renvoyer plusieurs résultats simultanément : il suffit de renvoyer chaque valeur séparée par une virgule, et affecter ce résultat à des variables de la même manière lors de l'appel à la fonction :

```
def min_et_max(li):
    mini, maxi = li[0], li[0]
    for elt in li:
        if mini > elt:
            mini = elt
        if maxi < elt:
            maxi = elt
    return mini, maxi
-------------------------------------------------------------------------
borne_inf, borne_sup = min_et_max(liste_fournie)
```

1.3. Syntaxe hors-programme : les annotations

Il existe une syntaxe possible pour indiquer aux lecteur·rices les types attendus pour les paramètres et le return : en utilisant les exemples précédents :

```
def somme_liste(li: list) -> float:
    # les instructions...
def remise_à_zéro(li: list) -> None:
    # les instructions...
def min_et_max(li: list) -> (float, float):
    # les instructions...
def exemple_général(x: type_de_x, y: type_de_y...) -> type_du_return:
    # les instructions...
```

Cette syntaxe n'est pas à connaître, mais elle est parfois utilisée dans les sujets pour aider à comprendre ce qui est attendu de la fonction. Il faut considérer que ces éléments **ne sont pas lus par Python lors de l'exécution** : ils sont finalement exactement équivalents à des commentaires qui indiqueraient le type attendu.

II. Gestion des paramètres

2.1. Gestion mémoire des paramètres lors de l'appel

a) Évaluation à l'appel

Prenons l'exemple de la fonction suivante[3] :

```
def somme_nulle(x, y, z):
    t = x + y + z
    return t
```

Que se passe-t-il en mémoire lors de l'appel suivant :

```
In [1]: res = somme_nulle(2, 3, -4)
```

Au début de l'exécution de la fonction, tout se passe comme si on créait trois nouvelles étiquettes `x`, `y` et `z` au début de la fonction et qu'on leur donnait les valeurs passées en paramètres ; autrement dit, le corps de la fonction se comporte comme

```
def somme_nulle_modifiée():
    x, y, z = 2, 3, -4
    t = x + y + z
    return t
```

Bien entendu, cette nouvelle version est (encore plus) inutile que `somme_nulle`, puisque pour l'appeler avec de nouvelles valeurs de paramètres, il faut la re-coder : mais elle présente la logique des étiquettes internes à une fonction, ainsi que celle du passage de paramètres.

[3] Certes inutile, c'est un exemple pédagogique simple...

L'évaluation de la valeur des paramètres se fait au moment de l'appel à la fonction, avant toute instruction dans celle-ci. Par exemple, on peut suivre le cas :

```
In [1]: liste = [1, 2, 3, 4, 5]
In [2]: var = -1
In [2]: res = somme_nulle(3 + 5, var, len(liste))
```

Dans ce cas, Python évalue les paramètres, et avant la première ligne de la fonction, il va

- voir `3 + 5` pour le premier paramètre, l'évaluer à 8, puis créer l'étiquette `x` et y affecter 8 ;
- voir `var` , regarder ce que vaut cette variable (ici -1), puis créer l'étiquette `y` et y affecter la valeur -1 ;
- voir `len(liste)` , appeler cette fonction ; le résultat étant 5, l'étiquette `z` est créée et la valeur 5 y est affectée.

Autrement dit, tout se passe comme si, juste avant la première ligne de la fonction, les instructions suivantes étaient exécutées :

```
x = 3 + 5
y = var
z = len(liste)
```

Ainsi, la fonction ne « connaît » pas l'origine des valeurs de `x` , `y` et `z` qui lui sont fournies : elles sont calculées en amont.

b) Variables mutables

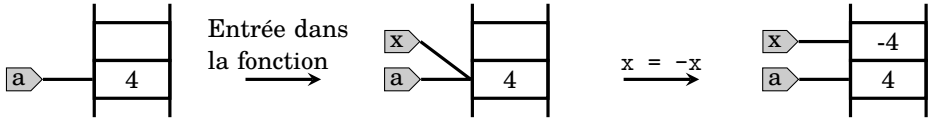
Un cas subtil est celui des variables mutables (listes pour l'instant) : nous avons vu que l'opération d'affectation créé un nouveau lien variable-étiquette, mais que l'on obtient ainsi plusieurs étiquettes sur une **même variable mutable**. Ainsi, si on prend les deux procédures suivantes :

```
def opposé_nombre(x):
    x = -x
def opposé_liste(li):
    n = len(li)
    for i in range(n):
        li[i] = -li[i]
```

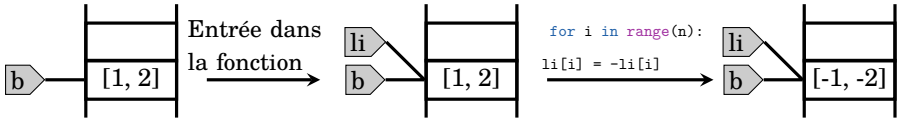
Leur comportement sera très différent : en effet,

```
In [1]: a, b = 4, [1, 2]
In [2]: opposé_nombre(a)
In [3]: print(a)
Out[1]: 4
In [2]: opposé_liste(b)
In [3]: print(b)
Out[1]: [-1, -2]
```

On peut comprendre la logique à l'aide des schémas ci-dessous : dans le cas de `opposé_nombre` , `x` est une étiquette qui pointe vers un nouvel entier au moment du changement de signe, donc l'étiquette `a` et la variable pointée ne sont pas impactées :



En revanche, une liste est un objet **mutable** en mémoire ! En représentant les listes comme un objet d'un seul bloc, le schéma devient le suivant :



Ces petits dessins avec des étiquettes en mémoire sont très utiles pour comprendre ces situations complexes.

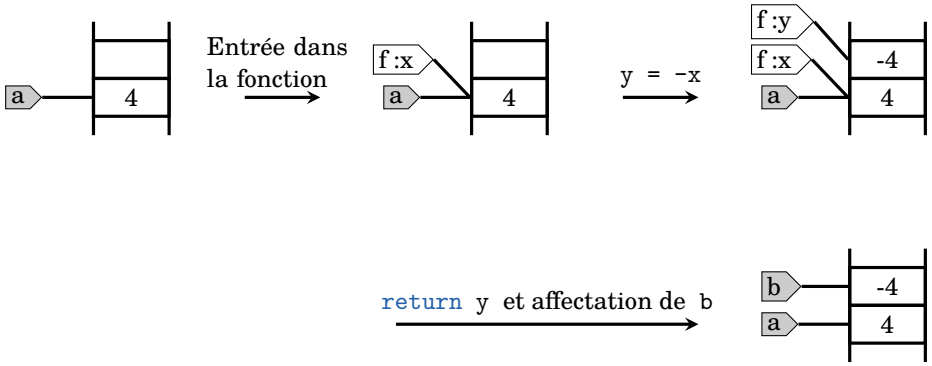
2.2. Portée des variables

a) Définition

Reprenons la fonction `opposé_nombre` dans une version modifiée et essayons de lire la valeur de `x` (définie dans le corps de la fonction, il s'agit du paramètre) une fois l'appel à celle-ci terminé :

```
def opposé_nombre(x):
    y = -x
    return y
-------------------------------------------------------------------------
In [1]: a = 4
In [2]: b = opposé_nombre(a)
In [3]: print(b)
Out[1]: -4
In [4]: print(x)
NameError: name 'x' is not defined
```

On constate que la variable `x` **n'existe plus** une fois la fonction terminée ; on obtiendrait la même erreur pour la variable `y` , définie à l'intérieur du corps de la fonction. En effet, **toute étiquette définie dans le corps d'une fonction est détruite à la sortie de celle-ci**. On dit que ces variables sont **locales** (y compris les paramètres). Pour reprendre l'exemple précédent, voici ce qui se passe en mémoire (avec les étiquettes locales ayant pour préfixe f et de couleur différente) :



Il est à noter que si les variables « internes » à une fonction sont qualifiées de locales, c'est parce que les variables « externes » à celles-ci sont qualifiées de **globales** : une variable globale est donc une variable qui existe dans le code en-dehors d'une fonction. Celles-ci peuvent être lues et modifiées par la fonction elle-même :

```
def fonction_inutile(x):
    return x + a
--------------------------------------------------------------------------------
In [1]: a = 2
In [2]: fonction_inutile(3)
Out[1]: 5
```

Ici, la fonction arrive à l'expression `x + a` et cherche à l'évaluer :

- elle cherche l'étiquette `x` dans l'espace local et obtient 3 ;
- elle cherche l'étiquette `a` dans l'espace local et n'en trouve pas ;
- avant de renvoyer une erreur, elle cherche l'étiquette `a` dans l'espace global et obtient 2 ;
- elle peut donc évaluer l'expression et la renvoyer.

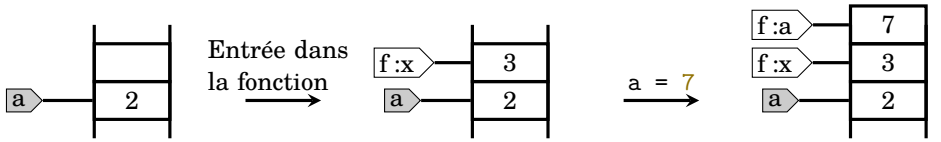
Si la variable `a` n'existait pas du tout, on aurait eu une erreur à l'exécution :

```
def fonction_inutile(x):
    return x + a
--------------------------------------------------------------------------------
In [1]: fonction_inutile(3)
NameError: name 'a' is not defined
```

et si `a` était redéfini dans le corps de la fonction, l'étiquette locale aurait eu la priorité

```
def fonction_inutile(x):
    a = 7
    return x + a
--------------------------------------------------------------------------------
In [1]: a = 2
In [2]: fonction_inutile(3)
Out[1]: 10
```

suivant le schéma suivant



> En général, utiliser ou modifier des variables globales depuis une fonction est vu comme une **mauvaise pratique** : cela rend le code plus difficile à lire, et la fonction moins souple (puisqu'on ne peut pas changer les paramètres aisément, ou l'utiliser sur plusieurs variables aux noms différents). Leur utilisation est justifiable pour des paramètres qui ne varient pas sur l'ensemble du code, par exemple la taille de l'échiquier dans un programme d'échecs, mais c'est un usage assez rare : dans le doute, donnez toutes les informations dont la fonction a besoin en paramètres.

Définition – portée des variables

Une variable définie dans le corps d'une fonction ou dans ses paramètres a une **portée locale**.

Une variable définie hors d'une fonction et avant son appel a une **portée globale**.

b) Cas des variables mutables

Le cas subtil est encore celui des listes : dans l'exemple de `opposé_liste` vu précédemment, l'étiquette `b` est globale et l'étiquette `li` est locale[4]. Cependant, les deux étiquettes référencent une **même liste**, qui est donc globale. Il faut être particulièrement attentif à ce genre de chose : si on demande de créer une copie d'une liste, par exemple,

```
def copie(li):
    cop = li
    return li
```

ne fait absolument pas le travail ! On a simplement créé des nouvelles étiquettes sur la **même liste**. Une solution possible est

```
def copie(li):
    cop = [x for x in li]
    return cop
```

qui créé effectivement une **nouvelle liste**, et lit chaque élément de `li` pour créer une nouvelle étiquette `cop[i]` qui pointe vers cet élément.

[4] Ainsi que `n` et `i`, mais ce ne sont pas des variables mutables

2.3. Cas particulier de paramètres : les fonctions

En Python, le nom d'une fonction est en fait une étiquette qui désigne cette fonction. Il s'ensuit qu'on peut **donner comme paramètre une fonction**.

> De la même manière qu'on peut voir une liste comme un objet Python pour qui l'opérateur d'indice `[]` est défini, on peut donc simplement voir une fonction comme un objet Python pour lequel l'opérateur `()` est défini.

Ceci permet de définir des algorithmes qui appellent d'autres fonctions. Par exemple, on peut vouloir calculer le zéro d'une fonction par la méthode de dichotomie. Pour n'écrire l'algorithme qu'une fois, on peut passer la fonction sur laquelle l'appliquer en paramètre :

```
def dichotomie(f, a0, b0, eps):
    a, b = min(a0, b0), max(a0, b0)
    fa = f(a)
    while b-a > eps:
        c = (a+b)/2
        fc = f(c)
        if fc*fa > 0:
            a = c
        else:
            b = c
    return (a+b)/2

# pour trouver un zéro de x -> exp(x) - 1/x**2
def fonction1(x):
    from math import exp
    return exp(x) - 1/x**2
# pour trouver un zéro de x -> sin(x) - ln(2+x/4)
def fonction1(x):
    from math import sin, log
    return sin(x) + log(2+x/4)
--------------------------------------------------------------------------------
In [1]: dichotomie(fonction1, 0.1, 1, 1e-6)
Out[1]: 0.703467
In [2]: dichotomie(fonction2, 0, 1, 1e-6)
Out[2]: 0.933096
```

2.4. Hors-programme : valeurs par défaut

On peut donner des valeurs par défaut aux paramètres : si on définition une fonction par `def fonction(x, y=0)` et qu'on l'appelle avec `fonction(3)`, ceci est équivalent à `fonction(3, 0)` (donc au début de l'exécution de la fonction, `x` vaut 3 et `y` vaut 0).

Cette possibilité est fort pratique mais n'est pas présente dans le programme officiel, donc évitez de l'utiliser au concours.

III. Contrat d'une fonction

3.1. Signature

Revenons sur la question du type des paramètres ou des valeurs de retour ; une réflexion sur ces types est capitale lorsqu'on pense la fonction. Prenons par exemple la fonction suivante :

```
def somme_liste(li):
    total = 0
    for elt in li:
        total += elt
    return total
```

Nous constatons que si on donne en paramètre un nombre, la ligne de la boucle `for` ne peut pas avoir de sens : cela conduit à une erreur

```
In [1]: somme_liste(4)
TypeError: 'int' object is not iterable
```

De même, si on donne en paramètre une chaîne de caractères, c'est cette fois la première occurrence de `total += elt` qui va poser problème, puisqu' `elt` sera du type `str` et `total` initialisé à 0 (donc `int`) :

```
In [1]: somme_liste('bonjour')
TypeError: unsupported operand type(s) for +=: 'int' and 'str'
```

De même, si on donne en paramètre une liste de nombres, on s'attend à ce que le résultat soit bien un nombre, pas autre chose ! Finalement, en rassemblant ces réflexions on, arrive à la description suivante de la fonction :

- son **nom** est `somme_liste` ;
- elle prend un paramètre qui doit être une liste de nombres ;
- elle renvoie un nombre.

L'ensemble de ces trois informations (nom, types des paramètres et du retour) est appelé **signature de la fonction** et est un élément fondamental de définition de la fonction. Du point de vue de l'utilisateur, peu importent les détails du traitement interne, une description de la signature et de l'utilité de la fonction est bien plus utile.

Il est vivement conseillé (et obligatoire dans le cadre du programme) de préciser la signature d'une fonction lorsqu'on l'écrit. Il y a plusieurs façons de faire :

- on peut simplement mettre ces précisions en commentaires

```
def somme_liste(li):
    # li doit être une liste de nombres ;
    # la fonction renvoie un nombre, somme des éléments de la liste
    ...
```

- on peut également utiliser les annotations

```
def somme_liste(li: list) -> int:
    ...
```

mais cette syntaxe est **hors-programme** : elle pourra donc être utilisée dans les sujets

ou les cours (parce qu'assez naturelle à lire), mais elle ne sera jamais exigible.
- on peut enfin placer la signature dans la docstring, ce qui concerne le prochain paragraphe.

3.2. Documentation

Rappelons-nous de l'instruction `help` : utilisée dans un terminal, elle permet d'obtenir une documentation

```
In [1]: help(len)
| Help on built-in function len in module builtins:
|
| len(obj, /)
|     Return the number of items in a container.
```

On peut obtenir le même résultat à l'aide de la **docstring** : il s'agit d'une chaîne de caractères (généralement délimitée avec les triples guillemets) placée juste après le nom de la fonction, avant toute autre instruction. Cette chaîne de caractère sera alors comprise par Python comme la **documentation** de la fonction :

```
def somme_liste(li):
    """
    Effectue et renvoie la somme des nombres contenus dans une liste.
    Args:
        li (list[int/float]): ensemble de nombres
    Returns:
        somme (int/float) : la somme des nombres de l'ensemble
    """
    # les_instructions...
```

```
In [1]: help(somme_liste)
| Help on function somme_liste in module __main__:
|
| somme_liste(li)
|     Effectue et renvoie la somme des nombres contenus dans une liste.
|     Args:
|         li (list[int/float]): ensemble de nombres
|     Returns:
|         somme (int/float): la somme des nombres de l'ensemble
```

Il est vivement recommandé de donner les spécifications d'une fonction (notamment sa signature, mais également une description succincte) en docstring systématiquement.

3.3. Préconditions et postconditions

On peut aller plus loin dans la spécification d'une fonction : on peut
- préciser les conditions nécessaires à son bon fonctionnement : on les appelle les **préconditions** ; les types des paramètres en font partie ;
- préciser ce que la fonction s'engage à fournir comme service si les préconditions sont remplies : on l'appelle la **postcondition** ; le type de la valeur de retour en fait partie.

L'ensemble préconditions-postcondition est appelé **contrat** de la fonction : en général, on l'inscrira dans la docstring. Détaillons quelques exemples.

```
def racine_carrée(nb):
    """
    Renvoie la racine carrée d'un nombre réel positif.
    Args:
        nb (int/float): nombre
    Returns:
        x (float): racine carrée de nb
    Préconditions:
        nb >= 0
    Postconditions:
        x >= 0
        x**2 = nb
    """
    return nb**0.5
```

```
def suite_fibonacci(n):
    """
    Calcule les n premiers termes de la suite de Fibonacci,
    initialisée à u_0 = u_1 = 1.
    Args:
        n (int): nombres de termes à calculer
    Returns:
        suite (list[int]) : les n premiers termes de la suite.
    Préconditions:
        n > 1
    Postconditions:
        len(suite) == n
        pour tout indice i > 1, suite[i] == suite[i-1] + suite[i-2]
    """
    suite = [1, 1]          # les rangs 0 et 1 sont initialisés
    for i in range(2, n):  # on démarre à 2
        suite.append(suite[i-1] + suite[i-2])
    return suite
```

Si une précondition n'est pas vérifiée, la fonction aura a priori un comportement indéfini. Plutôt que d'accepter un tel comportement, on peut vouloir tester une précondition et arrêter le programme si elle n'est pas remplie : c'est ce que permet le mot-clé `assert` : on évalue un booléen, et s'il est `False`, le programme est arrêté ; s'il est `True`, tout continue.

```
def racine_carrée(nb):
    """
    Renvoie la racine carrée d'un nombre réel positif.
    Args:
        nb (int/float): nombre
    Returns:
        x (float): racine carrée de nb
    Préconditions:
        nb >= 0
    Postconditions:
        x >= 0
        x**2 = nb
    """
    assert nb >= 0
    return nb**0.5
--------------------------------------------------------------------------------
In [1]: racine_carrée(-5)
AssertionError:
```

L'erreur n'est pas très explicite : on peut la préciser en fournissant une chaîne de caractères à `assert` en plus d'un booléen (sous forme de tuple `(bool, str)`) :

```
def racine_carrée(nb):
    """
    La docstring...
    """
    assert nb >= 0, "La fonction racine_carrée attend un nombre positif"
    return nb**0.5
--------------------------------------------------------------------------------
In [1]: racine_carrée(-5)
AssertionError: La fonction racine_carrée attend un nombre positif
```

3.4. Discipline de programmation : les tests

La notion de contrat nous amène à une question suivante : comment s'assurer que la fonction écrite respecte bien les postconditions ? Une bonne habitude (utilisée dans l'industrie) consiste à tester les fonctions. Pour ce faire, on choisit un certain nombre de valeurs des paramètres dont on connaît le résultat attendu, et on vérifie que cela marche avec `assert`. Par exemple,

```
In [1]: assert racine_carrée(0) == 0, "erreur de valeur pour sqrt(0)"
In [2]: assert racine_carrée(1) == 1, "erreur de valeur pour sqrt(1)"
In [3]: assert racine_carrée(4)**2 == 4, "erreur de valeur pour sqrt(4)"
In [4]: assert racine_carrée(1e6)**2 == 1e6, "erreur de valeur pour sqrt(1e6)"
```

Notons quelques idées :

- on essaie d'identifier quelques cas particuliers (ici 0 et 1) ;
- on teste des valeurs un peu réparties dans le domaine des entrées acceptables.

L'idéal n'est pas de faire chaque test à la main comme ici, mais plutôt de les regrouper dans une procédure, éventuellement en regroupant les valeurs.

```
def tests_racine_carrée():
    """
    Fonction des tests unitaires de racine_carrée.
    Postconditions testées :
        x**2 = nb
    On teste en particulier les cas 0 et 1.
    """
    phrase_erreur = "Test de racine_carrée échoué pour "
    for k in [0, 1]:
        assert racine_carrée(k) == k, phrase_erreur + "{:.4f}".format(k)
    for k in [4, 10, 1e6, 7e12]:
        assert racine_carrée(k)**2 == k, \
          phrase_erreur + "{:.4f}".format(k)
```

Ainsi, si la fonction ne passe pas les tests, on saura immédiatement quelle entrée pose problème ; on sera plus efficace dans le débuggage avec cette information.

En conditions réelles, il est recommandé d'écrire l'ensemble des tests (regroupés dans une procédure) avant même d'écrire la fonction, afin de s'assurer qu'on a bien cerné le cahier des charges que doit remplir la fonction. Dans chaque TD « réel »[5], il vous est fortement recommandé d'écrire quelques tests (deux ou trois) avant la fonction demandée.

> Dans les TD, les questions seront presque exclusivement formulées sous la forme « écrire une fonction/procédure qui... » ; cela ne vous empêche pas d'ajouter une fonction `test_nom_de_la_fonction` à chaque fois. Par ailleurs, il pourra m'arriver de faire moi-même des corrections par tests : si votre fichier passe les tests, vous avez les points.

> En situation industrielle, il existe des outils qui aident à générer automatiquement les tests pour gagner du temps ; cependant ces outils demandent **beaucoup** d'apprentissage et ont donc été exclus du programme. Seule la discipline de programmation et la réflexion associée aux tests sont attendues.

[5] C'est-à-dire pas les mini-exercices en parallèle du cours

ITC – cours n°5

Exercices

1 – Écrire une procédure `opposé_en_place` qui prend en paramètre une liste de nombre et change le signe de chaque nombre de la liste.

2 – Écrire une fonction `opposé_copie` qui prend en paramètre une liste de nombres `li` et renvoie une **nouvelle liste** `opp` telle que pour tout indice `i` valide de la liste, `opp[i] == -li[i]`

3 – Écrire un jeu de tests pour la fonction précédente. On vérifiera en particulier que la liste donnée en paramètre n'est pas copiée.

4 – Écrire une docstring pour la fonction précédente.

5 – On considère les trois fonctions suivantes :

```
def f1(li):
    li.apppend(3)
    return li
def f2(li):
    li = li + [3]
    return li
def f3(li):
    li = [x for x in li]
    li.append(3)
    return li
```

Laquelle modifie la liste passée en paramètre ? On justifiera en détail.

6 – On donne la fonction suivante

```
def doubler(li):
    n = len(li)
    for i in range(n):
        li[i] = 2*li[i]
```

que l'on utilise dans un terminal suivant les instructions :

```
In [1]: liste = [4, 5, 1, 3]
In [2]: liste = doubler(liste)
In [3]: liste[0] = 4
TypeError: 'NoneType' object is not subscriptable
```

Expliquer l'erreur et corriger le code.

7 – Proposer un jeu de tests pour la fonction `doubler` .

8 – On considère une liste Python contenant des listes Python : `li[i]` est donc une liste pour tout indice valide. On propose la fonction suivante pour faire une copie indépendante d'une telle liste :

```
def copie_liste_liste(li):
    cop = []
    n = len(li)
    for i in range(n):
        cop.append(li[i])
    return cop
```

Expliquer pourquoi cette fonction ne fait pas une copie indépendante. Proposer une version qui fonctionne.