Chapitre (S2) 1 Bases de programmation

Objectifs

- Savoir définir les spécifications (signature, docstring).
- Savoir annoter un bloc d'instructions (précondition, postcondition, invariant).
- Savoir mettre au point un jeu de tests.

1. FONCTIONS ET EFFET DE BORD

1.1. Copie de variables mutables et non mutables

Exercice 1 [Sol 1] On considère le script suivant qui met en jeu des copies soit d'entier, soit de liste :

```
x = 0
y = x
x = 1

L =[0]
Lp = L
L[0] = 1
```

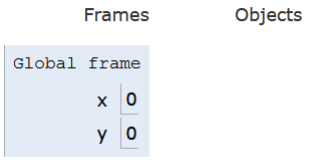
1) Pouvez-vous prévoir les valeurs associée aux variables `y` et `Lp` après exécution du script ?

Pour comprendre la différence de traitement de ces instructions qui semblent similaires, il faut préciser la gestion mémoire du contenu des variables dans le langage Python. Cette gestion est différente selon que la variable est *non mutable* (entiers, flottants par exemple) ou *mutable* (listes, dictionnaires, tableaux par exemple).

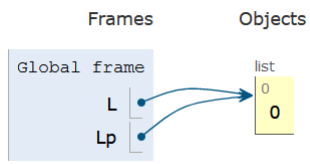
Plus précisément, une variable peut être vue comme une étiquette (un nom) donné à un emplacement mémoire. Mais le contenu de cet emplacement dépend du caractère mutable ou non de la variable :

- pour une variable non mutable, le nom désigne directement l'adresse mémoire de la valeur (entier, flottant ou chaine de caractère) associée à la variable,
- pour une variable mutable, le nom désigne un objet particulier, appelé pointeur, qui contient l'adresse mémoire de l'endroit où est stockée la valeur associée à la variable.

On peut alors représenter les deux situations de copie de variable vues dans l'exemple introductif de la manière suivante :



variable non mutable x

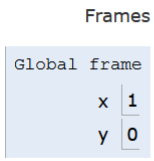


variable mutable L

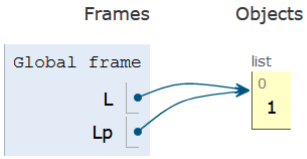
Lors d'une copie :

- pour un objet non mutable, par exemple pour l'instruction `y = x`, la variable `y` désigne une nouvelle case mémoire contenant la même valeur que celle de la variable `x` et cette valeur est alors dupliquée (elle est présente deux fois dans la mémoire),
- pour un objet mutable, par exemple pour l'instruction `Lp = L`, la variable `Lp` désigne une nouvelle case mémoire contenant un pointeur désignant lui-même le même emplacement mémoire que celui du pointeur de la variable `L`. Cette dernière valeur n'est alors **pas** dupliquée (elle n'est présente qu'une fois dans la mémoire).

Après copie et modification des valeurs de `x` et `L[0]`, on obtient donc la situation suivante :



après modification de `x`



après modification de `L`

Ceci permet d'expliquer le comportement observé, et cette différence de comportement peut se généraliser à tous les objets mutables et non mutable.

1.2. Effets de bord sur les fonctions

Le comportement précédent explique aussi la façon dont les fonctions agissent sur les variables mutables qui sont passées en paramètre. Ce comportement porte le nom *d'effet de bord* (ou *side effect* en anglais), que l'on peut définir comme suit :

Définition 1 | Fonction à effet de bord

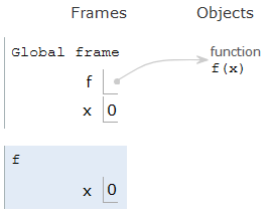
En informatique, une fonction est dite à *effet de bord* si elle modifie un état en dehors de son environnement local, c'est-à-dire a une interaction observable avec le monde extérieur autre que retourner une valeur.

Prenons l'exemple suivant :

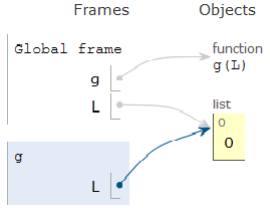
```
def f(x):
    x = 1
x = 0
f(x)

def g(L):
    L[0] = 1
L = [0]
g(L)
```

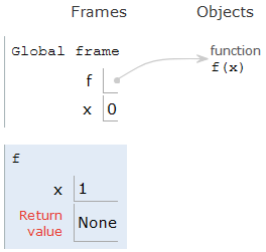
Après exécution, on observe que la valeur associée à `x` est 0, et que la valeur associée à `L` est `[1]`. Pour comprendre cette différence de comportement, on peut comme précédemment représenter le contenu des différentes variables au cours de l'exécution du script précédent :



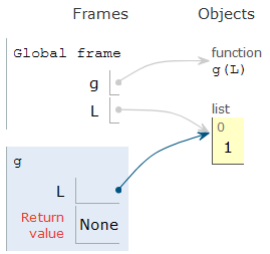
avant modif. de `x` par f



avant modif. de `L` par g



après modif. de `x` par f



après modif. de `L` par g

Lors de l'appel des fonctions `f` ou `g`, il y a création d'une variable locale (`x` ou `L`), qui n'a d'existence que durant l'exécution de la fonction (le fait que ces variables aient le même nom que les variables `x` et `L` du programme principal n'est pas gênant car les variables n'appartiennent en réalité pas au même espace des noms). On constate que pour la fonction `f`, qui agit sur la variable `x` non mutable, la variable locale `x` a dupliqué le contenu de la variable globale `x`, alors que pour la fonction `g`, la variable locale `L` pointe vers le même emplacement mémoire que celui la variable globale `L` (c'est le même comportement que lors de la copie de variable, vue précédemment). On comprend donc que la modification de `x` dans la fonction `f` n'affecte pas le contenu de la variable globale `x`, alors que la modification de `L` dans `g` affecte la variable globale `L`.

Ceci illustre deux aspects fondamentaux pour le comportement des fonctions :

- la façon dont une fonction modifie le contenu d'une variable d'entrée dépend du caractère mutable ou non de cette variable,
- il est possible de modifier la valeur associée à une variable d'entrée d'une fonction sans qu'il soit nécessaire pour la fonction d'avoir une valeur de retour. Ainsi :
  - pour modifier la valeur de la variable `x` non mutable dans le programme principal, il faut exécuter `x = f(x)`,
  - pour modifier la valeur de la variable `L` mutable dans le programme principal, il suffit d'exécuter `g(L)`,

Cette différence de comportement n'est pas gênante en soi à condition d'avoir bien conscience lors de la conception des fonctions.

1.3. Compléments sur l'affectation : expression vs instruction

Dans le langage Python, l'instruction d'affectation se note =, alors que le test d'égalité se note ==, et ces deux symboles n'ont pas le même statut vis à vis du langage. Plus précisément :

- le symbole = est une *instruction* du langage, c'est à dire un symbole réservé qui réalise une action et ne renvoie rien (comme `import`, `for`, `while`, `if`, `else`, etc.). On peut notamment signaler que la commande `print(a = 1)` n'est pas comprise par l'interpréteur, vu que `a = 1` ne renvoie rien, et que l'écriture de `if a = 1` provoque également une erreur (`SyntaxError`) lors de la compilation.
- le symbole ==, comme tout opérateur de test (>,>=,<,<=, !=), permet de construire une *expression* qui renvoie une valeur booléenne (`True` ou `False`). Ainsi, `print(a == 1)` renvoie `True` ou `False` (à condition que la variable `a` ait été définie précédemment).

Cette distinction peut paraître fastidieuse (et souvent source de confusion pour le néophyte), mais permet de différencier les deux opérations dans le langage, ce qui permet d'éviter des confusions susceptibles d'engendrer des erreurs. Ceci n'est pas le cas dans tous les langages. Par exemple en C, l'opération d'affection est une *expression* (et pas une instruction) qui renvoie la valeur que l'on cherche à affecter (`a=1` affecte la valeur 1 à `a` *et* renvoie 1). Ainsi la syntaxe `if a = 1` est acceptée par le compilateur C car `a = 1` renvoie la valeur 1, et en C , une des façons de coder les valeurs logiques `True` et `False` est d'utiliser 0 pour `False` et tout autre nombre pour `True` (cette possibilité existe aussi en Python mais comme on l'a vu, `if a = 1` renvoie `SyntaxError` en Python).

2. SPÉCIFICATIONS

2.1. Le but : exprimer un besoin

Un programme informatique est en général composé de fonctions, chacune étant conçue pour répondre à un problème donné, ce problème pouvant être issu de domaines très variés (mathématique, physique, industrie, etc.). Par exemple, on peut vouloir déterminer le plus court chemin entre deux points, résoudre numériquement une équation différentielle, ou bien contrôler la vitesse d'un avion en fonction des paramètres de vol.

La spécification est l'explication (en général sous la forme d'une description en langage courant) du rôle de la fonction, accompagnée de la description de ses paramètres d'entrée, et de ses valeurs de sortie. Cette spécification est indépendante de la méthode algorithmique mise en jeu pour répondre au problème posé. La présence de spécifications présente de nombreux avantages :

- meilleure compréhension du rôle de la fonction, sans qu'il y ait nécessité de lire le code informatique,
- meilleure compréhension de l'interaction entre fonctions au sein du même programme,
- relecture d'un ancien code plus facile,
- travail collaboratif entre plusieurs utilisateurs ou développeurs facilité.

2.2. Syntaxes

La spécification d'une fonction s'effectue à l'aide de précisions apportées dans le code, et qui on vocation à être lues par l'homme (contrairement au code informatique en tant que tel, qui lui est lu et interprété par la machine). On dispose pour cela essentiellement de deux outils, la *signature* d'une fonction, et/ou la *docstring* d'une fonction.

2.2.1. Signature

La signature est la précision des types des paramètres d'entrée et de la valeur de sortie de chaque fonction. Cette précision s'effectue lors de la définition de la fonction, selon la syntaxe générale :

```
def func(param1:type1, param2:type2, ...)->type_sortie:
```

où `param1, param2,...` sont les noms des paramètres d'entrée de la fonction, `type1, type2,...` les types des paramètres d'entrée, `type_sortie` le type le la valeur de sortie. Ces informations sont purement déclaratives et ne sont pas vérifiées par l'interpréteur au moment de l'exécution. Cependant les types doivent être des types connus du langage. On peut notamment utiliser :

- `int`
- `str`
- `dict`
- `float`
- `list`
- `np.array`

On peut aussi signaler le type **`None`**, qui sert à préciser la sortie d'une fonction sans valeur de retour (comme par exemple une fonction d'affichage, ou bien une fonction qui agit sur une liste par effet de bords uniquement). On aura par exemple les signatures suivantes :

- `f(a:int,y:float)->float`
- `f(L:list,v:int)->bool`
- `f(t:np.array,v:int)->None`

Enfin, pour des structures composées comme des listes (mais aussi des tableaux), on peut spécifier le type des éléments qui constituent la structure. Ainsi on pourra écrire :

- `f(L:[int])->int` (au lieu de `f(L:list)->int`) pour une fonction travaillant sur une liste d'entiers et renvoyant un entier,
- `f(d:dict)->[[int,str]]` pour une fonction renvoyant une liste de listes composées chacune d'un entier et d'une chaine de caractère, à partir d'un dictionnaire.

2.2.2. Docstring

La docstring (contraction de « documentation string ») est un texte descriptif écrit par les programmeurs principalement pour eux-mêmes dans le but d'expliquer le rôle d'une fonction, et de permettre une utilisation pertinente de celle-ci. Les informations présentes dans la docstring précisent le rôle et l'utilisation de la fonction, mais ne rentrent pas dans le détail du code (en cela elles diffèrent des commentaires, que nous verrons plus loin).

Le texte de la docstring figure juste après la définition de la fonction, et est délimité par des triples guillemets. Il est indenté comme le corps de la fonction. Les informations présentes dans la docstring d'une fonction `func` sont accessible par les expressions `func.__doc__` ou `help(func)`. Dans la communauté informatique, les règles d'usages pour l'écriture d'une docstring sont normées et il existe différentes normes (reStructuredText, Numpydoc, Googledoc), mais nous ne rentrerons pas dans ces raffinements. Nous pourrons par exemple utiliser un format de présentation proche de celui de Numpydoc.

Les principaux éléments qui doivent figurer sont les suivants :

- une description de l'action de la fonction
- une section Parameters (ou Paramètres), précisant les paramètres (ou variables) d'entrée, leur type et leur description (i.e. ce que représente chacun des paramètres d'entrée),
- une section Returns (ou Renvoi), précisant la variable de sortie, son type et sa description,
- une section Examples (ou Exemples), non obligatoire mais fortement recommandée, illustrant l'action de la fonction sur un jeu de variables d'entrée donné.

2.3. Exemples

2.3.1. Fonction d'addition

Voici le script d'une fonction `add` dont on a précisé à la fois la signature et la docstring.

```
def add(x:float,y:float)->float:
    """
    Calculate the sum of 'x' and 'y'

    Parameters
    ----------
    x : float
        first value to add
    y : float
        second value to add

    Returns
    -------
    float
        the sum of 'x' and 'y'

    Examples
    --------
    >>>add(1,2)
    3

    """
    return x+y
```

Sur cet exemple introductif relatif à une fonction « simple » , l'intérêt de la docstring peut paraître nul, mais il est cependant essentiel de prendre l'habitude de documenter ses fonctions (et plus largement ses programmes). On remarque également dans cet exemple que la documentation peut être plus longue que le code lui-même.

2.3.2. Usages respectifs

On voit que la docstring précise les types des variables d'entrée et le type de la variable renvoyée, elle contient donc toutes les informations présentes dans la signature. Il est donc superflu de préciser à la fois signature et docstring pour une fonction donnée (même si dans les premier exemples qui suivent, les deux sont indiqués, à titre d'entrainement). En pratique :

- dans le code informatique d'une fonction, on précisera toujours la dosctring, et la signature est alors superflue,
- quand on décrit une fonction sans en donner le code, le fait d'écrire la signature de la fonction permet de mieux appréhender le rôle de cette fonction. Par exemple pour une fonction `sum` qui renvoie la somme des éléments contenus dans une liste de flottants, l'écriture `sum(L)` est moins précise que l'écriture `sum(L:list)->float`.

2.4. Autres fonctions

On donne le code de la fonction `f` suivante :

```
def f(a,b):
    if b == 0 :
        return a
    return f(b, a%b)
```

On peut remarquer que :

- il n'est pas aisé de comprendre le rôle de cette fonction en première lecture,
- il est nécessaire de faire tourner l'algorithme à la main pour en comprendre le rôle.

Exercice 2 **Spécification** [Sol 2]

1) Appliquer la fonction à la main aux couples $(15,10)$ et $(35,21)$. Que semble faire cette fonction ?

2) Réécrire la fonction en précisant la signature et la docstring. Il pourra également être utile de renommer la fonction.

Exercice 3 **Recherche dichotomique** [Sol 3] On donne ici un script de recherche dichotomique dans une liste triée (par ordre croissant), sans spécification ni signature :

```
def dicho(L, v):
    i_deb = 0
    i_fin = len(L)
    trouve = False
    while not trouve and i_deb < i_fin:
        i_m = (i_deb + i_fin) // 2
        if L[i_m] == v:
            trouve = True
        elif L[i_m] < v:
            i_deb = i_m + 1
        else:
            i_fin = i_m - 1
    return trouve
```

Écrire cette fonction en précisant sa signature et sa spécification.

Exercice 4 **Tri à bulles** [Sol 4] On considère la fonction `tri_bulle` suivante qui s'applique à une liste (ou un tableau) d'entiers et qui trie la liste (en la modifiant par effet de bord).

```
def tri_bulle(L):
    n = len(L)
    t = n-1
    fini = False
    while t > 0 and not fini:
        fini = True
        for j in range(t):
            if L[j] > L[j+1]:
                L[j],L[j+1] = L[j+1],L[j]
                fini = False
        t -= 1
```

Écrire cette fonction en précisant sa signature et sa spécification.

2.5. Instruction assert

La spécification présente la nature et la signification des paramètres d'entrée, mais ne permet pas de s'assurer que lors de l'utilisation d'une fonction, les valeurs des paramètres fournis à la fonction soient « corrects » . Par exemple, pour la fonction racine carrée réelle qui s'applique uniquement sur des réels positifs ou nuls, l'appel de cette fonction sur un réel négatif peut poser problème. Une des façons de résoudre ce problème est de vérifier systématiquement les propriétés des variables d'entrée est d'utiliser l'instruction `assert`, dont voici la syntaxe :

`assert condition [,message]` où :

- `condition` est un test effectué sur une ou plusieurs variables d'entrées (exemple $x > 0$, $a < b$, etc.),
- `message`, dont le caractère est optionnel, est une chaine de caractère précisant la condition testée.

Lors de l'exécution, si la condition est vérifiée, on passe à la ligne suivante, et sinon, la fonction s'interrompt en renvoyant `AssertionError`, suivi éventuellement du message d'erreur. Cette vérification permet de s'assurer que les paramètres d'entrée sont bien de la forme attendue. Les instructions `assert` sont alors placées en tout début, avant de rentrer dans le corps de l'algorithme. Cette pratique permet de rendre les fonctions plus robustes vis-à-vis d'une mauvaise utilisation, et présente un intérêt quand les programmes sont destinés à être utilisés par des utilisateurs variés. La démarche s'inscrit dans le cadre plus général de la programmation défensive.

2.5.1. Exemple de la fonction racine

Prenons la fonction racine carrée, basée sur la méthode de Héron, qui s'appuie sur le fait que pour $x \geq 0$, la suite :

$$u_0 = x, \quad u_{n+1} = \frac{1}{2}\left(u_n + \frac{x}{u_n}\right) \quad \text{qui converge vers } \sqrt{x}..$$

```
def sqrt(x:float,eps)->float:
    """
    Returns the square root of 'x'

    Parameters
    ----------
    x : float, assumed positive

    eps : float, assumed strictly positive
        fix the precision of calculus

    Returns
    -------
    float
        square root of x

    Examples
    --------
    >>> sqrt(2,1e-2)
    1.4142156862745097

    """
    assert type(x) == float or type(x) == int
    assert x >=0, "la variable 'x' est négative"
    assert type(eps) == float
    assert eps >0
    u = x
    v = 0.5*(u+x/u)
    while abs(v-u) > eps:
        u,v = v,0.5*(v+x/v)
    return v
```

L'utilisation des instructions `assert` permet de s'assurer que `x` est bien un réel (de type `int` ou `float`) positif, et que `eps` est bien un réel (de type `float`) strictement positif. Un appel à `sqrt` sur des paramètres incorrects met en évidence le rôle de `assert`.

```
>>> sqrt(-2,1e-2)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "S2C1.py", line 24, in sqrt
    assert x >=0
AssertionError: la variable 'x' est négative
```

Sans l'instruction `assert x >=0`, on obtient pour un appel avec $x = -2$, une boucle infinie.

Exercice 5 **Recherche d'une sous chaine** [Sol 5] Lors de la recherche d'un mot dans un texte, contenus dans deux chaines de caractère, on a utilisé une fonction

`mot_en_place(mot:str,texte:str,i:int)->bool`, qui renvoie **`True`** si et seulement si, en notant m la longueur de `mot`, on a :

$$\texttt{mot[j] = texte[i+j]}, \forall\ \texttt{j} \in [\![0, m-1]\!],$$

et **`False`** sinon.

1) Déterminer les contraintes que doivent vérifier les variables d'entrée de cette fonction.

2) Écrire les lignes de code correspondantes en utilisant `assert`.

3. ANNOTATIONS D'UN BLOC D'INSTRUCTIONS

3.1. Commentaires

Un code non commenté est un code inexploitable ! Que ce soit pour d'autres lecteurs éventuels (travail collaboratif par exemple), ou que ce soit pour soi-même (maintenance d'un programme sur le long terme par exemple), le code doit être assorti de commentaires dont le rôle est en particulier d'éclairer le code (mais pas seulement, comme nous le verrons plus loin).

Un commentaire doit être précédé du caractère #, tout ce qui suit ce caractère jusqu'à la fin de la ligne sera ignoré par l'interpréteur.

Quelques règles du bon usage des commentaires (extraits du PEP8) :

- Commentaire en ligne : un commentaire en ligne est un commentaire sur la même ligne qu'une déclaration. Les commentaires en ligne doivent être séparés par au moins deux espaces de la déclaration. Ils doivent commencer par un # suivi d'un seul espace.
- Bloc de commentaires : ils s'appliquent généralement à une partie (ou à tout) du code qui les suit et sont indentés au même niveau que celui-ci. Chaque ligne du bloc commence par un # suivi d'un seul espace (sauf s'il s'agit de texte en retrait à l'intérieur du commentaire). Les paragraphes à l'intérieur d'un bloc sont séparés par une ligne contenant un seul #.
- Les commentaires qui contredisent le code sont pires que pas de commentaires du tout ! Ne pas oublier de mettre à jour les commentaires lorsque le code change ! Les commentaires doivent être des phrases complètes et **pertinentes**. Le premier mot doit être en majuscule, sauf s'il s'agit d'un identifiant qui commence avec une lettre minuscule (ne jamais modifier la casse des identifiants !).
- Les blocs de commentaires consistent généralement en un ou plusieurs paragraphes construits à partir de phrases complètes, chaque phrase se terminant par un point. Vous devez utiliser deux espaces après le point de fin de phrase sauf après la dernière phrase.

```
# On initialise deux variables x et y à 1. Puis on calcule la \
↪ somme de x et y. (bloc de commentaire)
x = 1  # Initialisation de x (commentaire en ligne)
y = 1
z = x+y
```

Mais aussi clair que puisse être un code, cela ne prouve pas forcément qu'il fait ce que l'on attend de lui. Deux questions se posent systématiquement :

1. Est-ce que le code se termine ?
2. Si oui, le résultat obtenu est-il celui attendu ?

Ces questions nous amènerons dans un autre chapitre à la notion de preuve d'algorithme

On peut introduire dès à présent des outils pouvant figurer dans le code sous forme de commentaire :

- Précondition
- Postcondition
- Invariant de boucles

3.2. Précondition

Définition 2 | Précondition
Une *précondition* est une propriété (P) qui doit être vérifiée avant l'exécution du code.

Considérons l'extrait de code S suivant, où a et b désignent deux entiers naturels :

```
x, y = a, b
while x != y:
    if x > y:
        x = x - y
    else:
        y = y - x
```

S est donc défini pour $(a,b) \in \mathbb{N}^2$. Examinons l'évolution des valeurs du couple de variables (x,y) lorsqu'on exécute S avec $a = 14$ et $b = 9$:

$$(14,9) \to (5,9) \to (5,4) \to (1,4) \to (1,3) \to (1,2) \to (1,1)$$

la boucle s'arrête et on obtient $x = y = 1$.

Deuxième exemple, on exécute S avec cette fois-ci $a = 14$ et $b = 0$:

$$(14,0) \to (14,0) \to (14,0) \cdots$$

la boucle est infinie, S ne se termine donc jamais dans ce cas.

On peut alors partitionner l'ensemble de départ de S ($\mathbb{N}^2$ ici) en deux parties :

- L'ensemble $\{(a,b) \in \mathbb{N}^2 \text{ / S se termine et renvoie le résultat attendu}\}$,
- et son complémentaire qui est constitué des couples $(a,b) \in \mathbb{N}^2$ tels que S ne se termine pas, ou bien se termine mais ne donne pas le résultat attendu.

La proposition permettant de décrire le premier ensemble s'appelle **une précondition** (propriété qui doit être vérifiée avant d'exécuter le code), dans notre exemple ce pourrait être la proposition (P) suivante : « $(a,b) \in \mathbb{N}^2,\ a > 0,\ b > 0$ ».

Cette précondition peut être annotée dans le code sous forme d'un commentaire de la manière suivante :

```
# Précondition (P): a et b sont des naturels strictement \
↪ positifs
x, y = a, b
while x != y:
    if x > y:
        x = x - y
    else:
        y = y - x
```

3.3. Postcondition

Caractériser les conditions qui font qu'un code S s'exécute normalement et donne le résultat voulu ne suffit pas, il faut également caractériser le résultat attendu.

Définition 3 | Postcondition
Une *Postcondition* est une propriété (Q) qui doit être vérifiée après l'exécution du code.

Si on reprend l'exemple précédent, la postcondition pourrait être la proposition (Q) suivante : « $x = y$ et $x = \text{pgcd}(a,b)$ ».

Cette postcondition peut être annotée également dans le code sous forme d'un commentaire de la manière suivante :

```
# Précondition (P): a et b sont des naturels strictement \
↪ positifs
x, y = a, b
while x != y:
    if x > y:
        x = x - y
    else:
        y = y - x
# Postcondition (Q): x = y et x = pgcd(a,b)
```

Lorsque le code S est le corps d'une fonction, alors la précondition et la postcondition peuvent être mentionnées dans le docstring de la fonction.

3.4. Notion d'invariant

Définition 4 | Invariant
Un *invariant* est une proposition qui reste vraie tout au long de l'exécution du code (ou d'une portion du code).

Cette notion d'invariant est utilisée notamment pour les preuves d'algorithmes.

Dans l'exemple précédent l'invariant est : « x et y sont des naturels non nuls, et $\text{pgcd}(x,y) = \text{pgcd}(a,b)$ ». Il faut en général prouver l'invariance : ici on a initialement $x = a$ et $y = b$, donc l'invariant est bien vérifié au départ (la précondition est supposée remplie). Si celui-ci est vérifié à la fin de l'itération numéro n et s'il y a une itération $n+1$, alors la propriété mathématique qui dit que $\text{pgcd}(x,y) = \text{pgcd}(x-y,y) = \text{pgcd}(x,y-x)$, montre que dans les deux cas, le pgcd entre les nouvelles valeurs de x et y est le même que le pgcd entre les valeurs précédentes, qui par hypothèse est le $\text{pgcd}(a,b)$, de plus, comme $x! = y$, dans les deux cas, les nouvelles valeurs de x et de y sont des naturels strictement positifs, ce qui prouve que l'invariant et encore vérifié à l'issue de l'itération $n+1$ (par récurrence sur n).

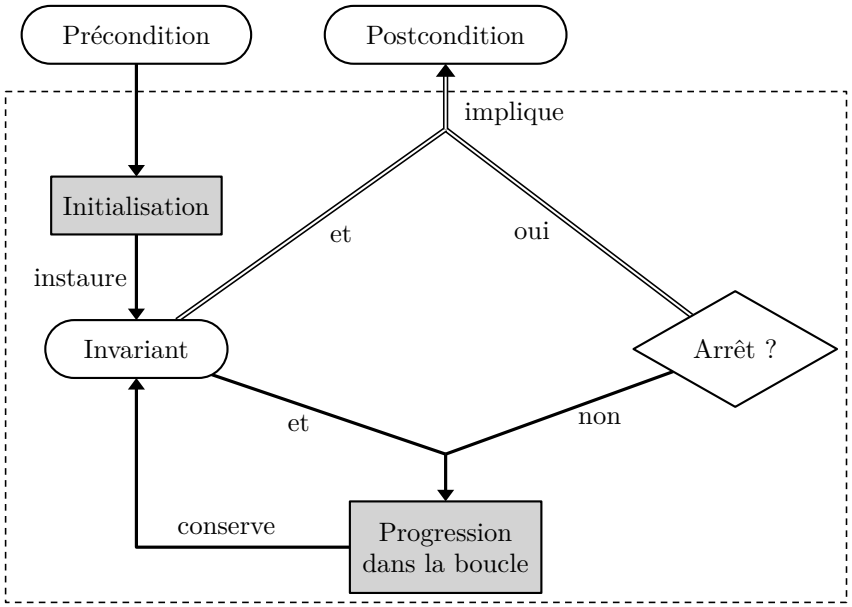
On peut alors noter cet invariant sous forme de commentaire dans le code, comme ceci :

```
# Précondition (P): a et b sont des naturels strictement \
↪ positifs
x, y = a, b
while x != y:
# Invariant: x et y sont des naturels strictement positifs
# et pgcd(x,y) = pgcd(a,b).
    if x > y:
        x = x - y  # pgcd(x, y) = pgcd(x - y, y)
    else:
        y = y - x  # pgcd(x, y) = pgcd(x, y - x)
# Postcondition (Q): x = y et x = pgcd(a,b)
```

Lorsque la condition n'est plus remplie, c'est à dire lorsque x et y sont égaux, on sort de la boucle, on a alors $x = y$ et d'après l'invariant on a aussi $\text{pgcd}(x, y) = \text{pgcd}(a, b)$, or $\text{pgcd}(x, y) = \text{pgcd}(x, x) = x$ (x est un naturel non nul), ce qui prouve la postcondition (Q).

On remarquera que nous n'avons pas prouvé que la boucle se termine forcément. Nous verrons les preuves de terminaison dans un autre chapitre. Pour une boucle, l'invariant doit être vérifié avant celle-ci (itération dite n° 0), et doit être vérifié également **à la fin** de chaque itération.

En résumé, lorsque nous avons affaire à une boucle, nous pouvons représenter la situation ainsi :



Les cases grisées correspondent à une action.

Exercice 6 [Sol 6] Écrire la fonction `somme(n: int) -> int` respectant la spécification suivante :

- Précondition : $n \in \mathbb{N}$.
- Postcondition : la valeur renvoyée est $\sum_{k=0}^{n} k$.

On précisera dans le code un invariant de la boucle.

Exercice 7 [Sol 7] On considère la fonction (non documentée) suivante :

```
def f(a, b):
    x, y, r = a, b, 0
    while x != 0:
        if x%2 == 0:
            x = x//2
        else:
            x = (x-1)//2
            r = r + y
        y = 2*y
    return r
```

1) Compte tenu des opérations effectuées il faut que a et b soient des nombres, et compte tenu de la division euclidienne faite sur x, il faut que a soit entier au départ. Que se passe-t-il lorsque $a = -1$? On pourrait vérifier qu'il se passe la même chose pour tout entier $a < 0$.

2) Montrer que la proposition : « $ab = xy + r$ » est un invariant pour la boucle `while`.

3) Réécrire le code cette fonction mais sous forme **documentée**.

4. JEUX DE TESTS

4.1. Utilité

Du point de vue purement intellectuel, une fois qu'un programme est clairement spécifié et prouvé, il peut sembler inutile de le tester. Sauf que nous ne sommes pas

infaillibles dans nos démarches intellectuelles et que des erreurs peuvent se glisser malgré tout. D'où la nécessité de multiplier les tests, que ce soit au niveau de chaque fonction écrite, ou au niveau de l'ensemble du programme (articulation entre les différentes parties du programme).

Contrairement à ce que l'on pourrait peut être penser, un test ne prouve pas le code, mais pour reprendre la citation d'Edsger DIJKSTRA : « *le test de programmes peut être une façon très efficace de montrer la présence de bugs, mais il est désespérément inadéquat pour prouver leur absence* » .

QUAND PRÉVOIR UN JEU DE TESTS ? Lorsqu'on doit écrire une fonction, il est conseillé en général de prévoir un jeu de tests dès la spécification de la fonction, donc bien avant l'écriture du code lui-même ! Un test est en fait la donnée de valeurs particulières pour les paramètres d'entrée de la fonction, et la valeur attendue en retour. Comme nous l'avons vu plus haut, l'ensemble des valeurs possibles des paramètres se partitionne en (au moins) deux parties, celle qui correspond aux valeurs pour lesquelles le code se termine et donne le résultat attendu, et la partie complémentaire, c'est en général sur les cas limites que portent les tests.

4.2. Exemple

On veut écrire la fonction `division(a, b)` dont la spécification est la suivante :

- Paramètres d'entrée : a et b sont des entiers positifs avec b non nul.
- Résultat renvoyé : le résultat de la fonction est le tuple (q, r) où q et r sont respectivement le quotient et le reste de la division euclidienne de a par b (c'est à dire vérifiant $a = bq + r$ avec $0 \leqslant r < b$).

Avant décrire le code documenté, il nous faut décider de ce que l'on va faire si la contrainte (ou précondition) sur les paramètres d'entrée n'est pas remplie, et prévoir un jeu de tests.

Pour la pré-condition non remplie, il y a plusieurs choix possibles :

- générer une erreur avec l'instruction `assert`,
- renvoyer un résultat particulier.

La première solution provoquera la fin du programme avec éventuellement un message d'erreur, ce qui n'est pas toujours souhaitable. Nous allons opter pour la seconde solution en convenant d'un résultat égal à `None` si la précondition n'est pas remplie. Cette convention devra être mentionnée dans la docstring.

Nous allons maintenant prévoir des tests pour les différents cas de figure, suivant que a est positif ou négatif, et b est nul ou strictement négatif ou strictement positif, en choisissant des valeurs pour a et b et en donnant le résultat qui devrait être renvoyé par la fonction, par exemple :

- a et b strictement positifs : `division(19, 7)` doit renvoyer `(2, 5)`,
- a et b strictement positifs : `division(7, 19)` doit renvoyer `(0, 7)`,
- a nul et b strictement positif : `division(0, 19)` doit renvoyer `(0, 0)`,
- a positif et b nul : `division(19, 0)` doit renvoyer `None`,
- a positif et b négatif : `division(19, -7)` doit renvoyer `None`,
- a négatif et b positif : `division(-19, 7)` doit renvoyer `None`,
- a négatif et b nul : `division(-19, 0)` doit renvoyer `None`,
- a négatif et b négatif : `division(-19, -7)` doit renvoyer `None`.

On pourrait bien sûr imaginer d'autres valeurs numériques, ou même choisir des valeurs aléatoirement dans chaque cas. Nous allons faire figurer ces test dans la docstring sous forme d'exemples

```
def division(a: int, b: int) -> (int, int):
    """
    Renvoie le quotient et le reste de la division de a par b

    Paramètres:
    ----------
        a: int, entier naturel
        b: int, entier strictement positif

    Retour:
    ------
        tuple (q, r) tel que a=bq+r avec 0<=r<b
        ou None si a<0 ou b<=0
    Exemples:
    --------
    >>> division(19,7)
    (2,5)
    >>> division(7,19)
    (0,7)
    >>> division(0,19)
    (0,0)
    >>> division(19,0)
    None
    >>> division(19,-7)
```

```
    None
    >>> division(-19,7)
    None
    >>> division(-19,0)
    None
    >>> division(-19,-7)
    None

    """
```

Nous pouvons maintenant passer à l'écriture du code. La post-condition nous fournit pratiquement l'invariant de la boucle que nous allons devoir écrire : $a = bq + r$, q et r seront deux variables locales, il faut les initialiser de sorte que l'invariant soit vérifié, il suffit de prendre $q = 0$ et $r = a$, la pré-condition nous dit alors que $0 \leqslant r$. Si $r < b$, c'est terminé, mais si $r \geqslant b$, alors l'idée est d'enlever b à r et d'ajouter 1 à q car $bq + r = b(q+1) + (r-b)$ (l'invariant est bien conservé), et on recommence le test sur r (boucle).

```
def division(a: int, b: int) -> (int, int):
    """
    Renvoie le quotient et le reste de la division de a par b

    Paramètres:
    ----------
        a: int, entier natuel
        b: int, entier strictement positif

    Retour:
    ------
        tuple (q, r) tel que a=bq+r avec 0<=r<b
        ou None si a<0 ou b<=0

    Exemples:
    --------
    >>> division(19,7)
    (2,5)
    >>> division(7,19)
    (0,7)
    >>> division(0,19)
    (0,0)
    >>> division(19,0)
    None
    >>> division(19,-7)
    None
    >>> division(-19,7)
    None,
    >>> division(-19,0)
    None
    >>> division(-19,-7)
    None

    """
    if (a<0) or (b<=0):  # pré-condition non remplie
        return None
    q, r = 0, a
    while r >= b:
    # Invariant: a=bq+r et 0<=r
        q += 1
        r -= b  # bq+r = b(q+1)+(r-b)
    return (q, r)
```

4.3. Comment effectuer les tests prévus ?

Une fois le code saisi et enregistré dans un fichier, on effectue les tests prévus. Nous allons donner trois façons de procéder :

- **[La méthode naïve]** on ajoute à la suite de notre fonction une succession de `print` (un par test), du style : `print(division(19,7) == (2,5))`, ce qui provoquera à l'exécution l'affichage de **`True`** ou bien **`False`** suivant que le test est positif ou négatif.
- **[Un peu plus élaboré]** on écrit une fonction dédiée aux tests qui va utiliser l'instruction `assert` pour chacun des tests :

```
def test_division():
    assert division(19,7) == (2,5), "erreur lorsque a=19 et b=7"
    assert division(7,19) == (0,7), "erreur lorsque a=7 et b=19"
    # ... etc
    assert division(-19,-7) == None, "erreur lorsque a=-19 et
b=-7"
    print("Tous les tests ont été réussis.")
```

Il n'y a plus alors qu'à exécuter cette fonction, s'il n'y a pas d'erreur d'assertion, on affiche que tous les tests ont été passés avec succès. Attention cependant, si un

des tests provoque une boucle infinie le programme ne se terminera pas, et on ne saura pas quel est le test défectueux.

- **[Tests automatiques]** lorsque les tests sont explicités dans la docstring sous une certaine forme, alors on peut faire passer automatiquement ceux-ci en utilisant le module `doctest` et plus précisément sa fonction `testmode()`, qui va tester l'ensemble des fonctions du programme. Pour chaque fonction, sa docstring va être analysée [1] et les lignes commençant par `>>>` (trois chevrons suivis d'une espace obligatoire) vont être considérées comme des instructions et être exécutées, le résultat de cette exécution est comparé avec ce qui est lu dans la ligne suivante, si cela ne coïncide pas une erreur est signalée. Si aucune erreur n'est signalée c'est que tous les tests ont été passés avec succès.
  Attention : la docstring est une chaîne de caractères, il faut donc faire très attention à la façon dont l'on écrit les résultats attendus car ce sont des chaînes de caractères qui vont être comparées. Par exemple si on écrit dans la docstring de notre fonction :

```
def division(a: int, b: int) -> (int, int):
    """
    ...
    >>> division(19,7)
    (2,5)
    ...
    """
    # ...
```

alors à l'exécution de l'instruction `doctest.testmod()` nous verrons l'erreur suivante :

```
**********************************************************************
File "val.py", line 250, in __main__.division
Failed example:
    division(19,7)
Expected:
    (2,5)
Got:
    (2, 5)
```

notez l'espace manquante après la virgule dans la docstring... De même, si on écrit :

```
def division(a: int, b: int) -> (int, int):
    """
    ...
    >>> division(19,-7)
    None
    ...
    """
    # ...
```

alors à l'exécution de l'instruction `doctest.testmod()` nous verrons l'erreur suivante :

```
**********************************************************************
File "val.py", line 264, in __main__.division
Failed example:
    division(-19,-7)
Expected:
    None
Got nothing
```

car `None` et `"None"` ce n'est pas la même chose ! Il est préférable d'opter pour l'écriture suivante :

```
def division(a: int, b: int) -> (int, int):
    """
    ...
    >>> division(19,-7) == None
    True
    ...
    """
    # ...
```

alors à l'exécution de l'instruction `doctest.testmod()` il n'y aura plus d'erreur (à condition d'écrire `True` correctement, et sans espace avant ni après !).

Pour conclure, nous pouvons proposer ce code pour tester notre fonction :

```
import doctest
def division(a: int, b: int) -> (int, int):
    """"
    Renvoie le quotient et le reste de la division de a par b

    Paramètres:
    ----------
        a: int, entier natuel
        b: int, entier strictement positif

    Retour:
    ------
        tuple (q, r) tel que a=bq+r avec 0<=r<b
        ou None si a<0 ou b<=0
```

1. on dit aussi « parsée », anglicisme issu du verbe to parse

```
    Exemples:
    --------
    >>> division(19,7) == (2,5)
    True
    >>> division(7,19) == (0,7)
    True
    >>> division(0,19) == (0,0)
    True
    >>> division(19,0) == None
    True
    >>> division(19,-7) == None
    True
    >>> division(-19,7) == None
    True
    >>> division(-19,0) == None
    True
    >>> division(-19,-7) == None
    True

    """
    if (a<0) or (b<=0):  # pré-condition non remplie
        return None
    q, r = 0, a
    while r >= b:
    # Invariant: a=bq+r et 0<=r
        q += 1
        r -= b  # bq+r = b(q+1)+(r-b)
    return (q, r)

doctest.testmod()
```

L'exécution ne provoque aucun affichage d'erreur ce qui signifie que tous les tests ont été passés avec succès. Cet exemple sera repris en TP pour l'étendre à la division dans $\mathbb{Z}$.

Remarque 1

- Le module `doctest` contient d'autres fonctions, par exemple il est possible de tester une fonction individuellement dans le programme (fonction `run_docstring_examples`).
- Il existe d'autres modules permettant des tests plus sophistiqués, mais nous n'en parlerons pas dans ce cours.

4.4. Tests de performance

Pour comparer des algorithmes, on peut être amené à effectuer d'autres types de tests, comme des tests de performance en temps d'exécution par exemple. Avec le module `time`, il est possible de faire ces mesures. On relève un instant initial, on exécute un certain nombre de fois la fonction, on relève l'instant final et il n'y a plus qu'à faire la différence :

```
from time import time  # fonction time du module time
t1 = time()  # instant initial
for _ in range(1000):  # pour 1000 exécutions
    r = fonction_a_tester()
t2 = time()  # instant final
print("durée: ", (t2-t1)/1000)  # en secondes
```

Suivant la précision de la machine, une seule exécution n'est peut-être pas suffisante pour avoir une mesure fiable.

Si on travaille dans un notebook, alors on peut plus simplement utiliser l'instruction `timeit fonction_a_tester()` qui va mesurer automatiquement le temps d'exécution de la fonction.

SOLUTIONS DES EXERCICES

Solution 1

1) La variable y contient la valeur 0 et la variable Lp contient la liste [1].

Solution 2 [ref=pgcd]

1)
- Pour les valeurs initiales $a = 15, b = 10$,on obtient successivement pour le couple (a,b) : $(10,5)$, $(5,0)$. La valeur retournée est donc 5.
- Pour les valeurs initiales $a = 35, b = 21$,on obtient successivement pour le couple (a,b) : $(21,14)$, $(14,7)$, $(7,0)$. La valeur retournée est donc 7.

Dans ces deux exemples, la fonction renvoie le pgcd (plus grand commun diviseur) de a, b.

2) Fonction avec spécifications :

```
def pgcd(a:int,b:int)->int:
    """
    Returns the pgcd of 'a' and 'b'

    Parameters
    ----------
    a : int
        first value to add
    b : int
        second value to add

    Returns
    -------
    int
        pgcd of 'a' and 'b'

    Examples
    --------
    >>>pgcd(15,10)
    5
    >>>pgcd(35,21)
    7
    """
    if b == 0 :
        return a
    return f(b, a%b)
```

Solution 3 [ref=dicho] En rajoutant les éléments demandés, on peut proposer pour la partie demandée (le code étant inchangé) :

```
def dicho(L:list, v:int)->bool:
    """
    Determines if value 'v' belongs to 'L' by dichotomic method.

    Parameters
    ----------
    L : list
        list of values sorted with L[0]<= L[1] <= ... <= L[n-1]
    v : int
        value to be tested

    Returns
    -------
    bool
        True if value 'v' is in 'L', False in the other case

    Examples
    --------
    >>>dicho([1,3,5],5)
    True
    >>>dicho([1,3,5],4)
    False
    >>>dicho([],1)
    False

    """
```

Solution 4 [ref=triBulle] On réécrit de même :

```
def tri_bulle(L:list)->None:
    """
```

```
    Sort the list in ascending order and return None. After \
    ↪ execution, the list is sorted in ascending order.

    Parameters
    ----------
    L : list
        list to be sorted

    Returns
    -------
    None

    Examples
    --------
    >>>L = [2,3,1]
    >>>tri_bulle(L)
    >>>L
    [1,2,3]

    """
```

Solution 5 [ref=recherche]

1) On peut lister plusieurs conditions :

- `mot` et `texte` doivent être des chaines de caractère,
- `i` doit être un entier positif ou nul,
- on doit avoir $i + \text{len(mot)} \leq \text{len(texte)}$.

2) Les lignes correspondantes s'écrivent :

```
    n = len(texte)
    m = len(mot)
    assert type(mot) == str
    assert type(texte) == str
    assert type(i) == int
    assert i >= 0
    assert i+m <= n
```

Solution 6 Somme des entiers (pour l'invariant on convient que la valeur de k avant la boucle est nulle) :

```
def somme(n: int) -> int:
    """
    Returns 0+1+...+n

    Parameters
    ----------
    n: int assumed positive

    Returns
    -------
    float
        0+1+...+n

    Examples
    --------
    >>> somme(10)
    55

    """
    resultat = 0  # variable qui contiendra la somme cherchée
    for k in range(1,n+1):  # pour k allant de 1 à n
    # Invariant : resultat est la somme des entiers de 0 à k
        resultat = resultat + k
    return resultat
```

Solution 7

1) Lorsque $a = -1$, la suite des valeurs de x est constamment égale à -1, la boucle est donc infinie.

2) Avant la boucle on a $x = a$, $y = b$ et $r = 0$, on a donc bien $ab = xy + r$. Supposons que cela soit vrai après l'itération n et qu'il y ait une itération $n+1$, on a donc $ab + xy + r$, distinguons deux cas :

- si x est pair alors $x//2 = \dfrac{x}{2}$ et $ab = xy + r = \dfrac{x}{2} \times (2y) + r$, on a bien la relation avec les nouvelles valeurs de x et de y (r n'a pas changé),
- si x est impair alors $x//2 = \dfrac{x-1}{2}$ et $ab = xy + r = (2\dfrac{x-1}{2} + 1)y + r = \dfrac{x-1}{2} \times (2y) + (y + r)$, on a bien la relation avec les nouvelles valeurs de x, de y et de r,

donc à l'issue de l'itération $n+1$ la relation $ab = xy + r$ est encore vérifiée.

3) Si la boucle se termine, alors x est nul et la valeur renvoyée est $r = r + xy = ab$ (car $x = 0$). D'où une documentation possible :

```
def f(a: int, b: float) -> float:
    """
    Returns a*b

    Parameters
    ----------
    a: int assumed positive
    b: float

    Returns
    -------
    float
        product a*b

    Examples
    --------
    >>> f(5,1.5)
    7.5

    """
    x, y, r = a, b, 0
    while x != 0:
    # Invariant: x*y+r = a*b
        if x%2 == 0: # x est pair
            x = x//2
        else:  # x est impair
            x = (x-1)//2
            r = r + y
        y = 2*y
    return r
```

Cette fonction effectue donc le produit d'un entier a avec un nombre b en utilisant uniquement des additions, soustractions, ainsi que des divisions et multiplications par 2 (multiplication russe). Il serait également judicieux de changer le nom de cette fonction pour quelque chose de plus explicite (par ex. `multiplication_russe`).