Manipulations des branchements et des boucles
Corrections des algorithmes
Variants et invariants de boucles

L'essentiel du cours

Voici un catalogue non exhaustif des syntaxes constamment utiles en PYTHON :

- opérateurs $+$, $-$, $*$, $/$, $**$, $//$, $\%$: addition, soustraction, multiplication, division, exponentiation, quotient dans la division euclidienne, reste dans la division euclidienne
- opérateurs de comparaison $==$, $!=$, $<=$, $>$: égalité, différence, inférieur ou égal, strictement supérieur
- affectation $=$: la syntaxe `x=a` affecte à la mémoire `x` la valeur de l'objet `a`.

1 Le branchement if

La syntaxe classique est :

```
if  condition 1

    bloc instructions 1

elif condition 2 :

    bloc instructions  2


elif condition k :

    bloc instructions k

else :

    dernier bloc instructions
```

Si l'une des conditions *condition* 1, *condition* 2, ..., *condition* k est réalisée, en notant i l'entier correspondant à la première condition réalisée, alors c'est le $i^{ème}$ bloc d'instructions seulement qui est exécuté.
Dans le cas contraire, c'est le dernier bloc d'instructions qui est réalisé.
En l'absence de else, rien n'est réalisé dans ce cas.
Conseil : énoncer des conditions disjointes pour faciliter la lecture du script.

2 La boucle for

La syntaxe classique est :

```
for x in L  :

    bloc instructions
```

Lorsque la variable x parcourt la liste L, le bloc d'instructions est exécuté. Celui-ci est donc exécuté $len(L)$ fois.
Par exemple, si $a \leqslant b$ sont deux entiers naturels, la syntaxe :

```
for k in range(a,b)  :

    bloc instructions
```

donne une répétition de $b-a$ fois du bloc d'instructions.
On rappelle que la syntaxe $range(a,b)$ modélise l'ensemble $[\![a,b[\![$ et que la syntaxe $range(n)$ modélise l'ensemble $[\![0,n-1]\!]$.
Conseil : ne pas modifier la liste L dans une boucle for.

3 La boucle while

La syntaxe classique est :

```
while condition :

    bloc instructions
```

Tant que la condition est vérifiée, on réalise le bloc d'instructions.
Le bloc d'instructions doit modifier des variables apparaissant dans la condition. Dans le cas contraire, la valeur booléenne de *condition* n'étant jamais modifiée, si elle est vraie au départ, elle le reste toujours et la boucle while plante.
Une boucle for peut toujours être remplacée par une boucle while :

```
k=0

while k<len(L)   :

    x=L[k]

    [reste du bloc instrutions]

    k+=1
```

remplace la syntaxe for x in L.
L'instruction break permet de sortir d'un bloc d'instructions. Cela peut être utile dans la syntaxe suivante :

```


while 1 :

    bloc instructions

    if condition :
        break
```

En PYTHON, les conditions booléennes peuvent être remplacées par des nombres. Tout nombre non nul fait office de True et le nombre nul fait office de FALSE.

4 Principes d'une correction partielle ou totale d'un algorithme

Le but de la suite du chapitre concerne la ***correction totale des algorithmes***, c'est-à-dire répondre aux deux questions suivantes :

- si l'algorithme fait intervenir une boucle while, ***cette boucle*** while ***se termine-t-elle en temps fini ?*** : il s'agit de démontrer ici la ***terminaison de l'algorithme*** où nous allons utiliser des variants de boucle
- ***le résultat renvoyé par l'algorithme est-il conforme à la solution de notre problème ?*** : il s'agit de démontrer ici la ***correction partielle*** de l'algorithme où nous allons utiliser des invariants de boucle
- la ***correction totale d'un algorithme*** revient à effectuer deux choses : prouver que les boucles while éventuellement présentes dans l'algorithme finissent après un nombre fini d'itérations et prouver ensuite que le résultat renvoyé par la procédure répond au problème posé au départ.

5 Les terminaisons de boucles

5.1 Les variants de boucle : définition

Considérons une boucle while du type :

```
while  condition C :

    bloc instructions B
```

Un ***variant de la boucle*** while ou encore une ***fonction de terminaison*** est une fonction T dépendant des paramètres mis en jeu dans le script et qui vérifie les conditions suivantes :

- la fonction T prend uniquement des valeurs entières
- à chaque réalisation du bloc d'instructions $\mathscr{B}$, la valeur obtenue pour la fonction T diminue strictement
- la condition « $T \leqslant 0$ » marque la sortie de boucle. Autrement dit, on a l'implication :

$$\text{la condition } \mathscr{C} \text{ est vraie } \iff T > 0.$$

Un variant de boucle T démontre que la boucle while se termine en temps fini. Le bloc d'instructions $\mathscr{B}$ ne se réalise qu'un nombre fini de fois. En effet, on utilise ici très fortement la propriété suivante :

Toute suite $(u_n)_{n\in\mathbb{N}}$ à valeurs entières et strictement décroissante prend des valeurs strictement négatives à partir d'un certain rang – et même diverge vers $-\infty$.

Un variant de boucle est totalement inutile dans une boucle for, à moins que celle-ci ne soit pas correctement implémentée. Une boucle for correctement implémentée termine toujours en temps fini : le nombre d'éléments de la liste dans laquelle varie la variable considérée.

5.2 Exemples d'application des variants de boucles

5.2.1 Exemple 1 : algorithme de recherche dans une liste

Considérons la fonction suivante :

```
def Appartient(x,L) :
    i=0
    trouve=False
    while i<len(L) and not(trouve) :
        trouve=L[i]==x
        i+=1
    return trouve
```

On « voit » que la boucle while va terminer en temps fini. Pour le formaliser davantage, on peut mettre en place le variant de boucle :

$$T = len(L) - i.$$

En effet, la quantité T est toujours un entier.
À chaque réalisation du bloc d'instructions, comme l'entier i est incrémenté d'une unité, alors la quantité T diminue strictement.
Enfin, la condition $T \leqslant 0$ devient $len(L) \leqslant i$ marquant la sortie de la boucle while.

5.2.2 Exemple 2 : algorithme du tri à bulles

On rappelle l'algorithme du tri à bulles :
$\longrightarrow$ partir d'une liste $L = [a_0, \cdots, a_{n-1}]$ de nombre réels

$\longrightarrow$ lire la liste L et à chaque fois que l'on trouve deux occurrences consécutives a_i et a_{i+1} telles que $a_i > a_{i+1}$, alors on les échange

$\longrightarrow$ recommencer l'étape précédente tant que de tels indices existent.

Il est ici plus difficile de voir directement que l'algorithme termine en temps fini.
On note $T = len(L) - s$, où s est le nombre de fois où la lecture complète de la liste L a été effectuée. En quelque sorte, l'entier s compte le nombre de fois où le bloc d'instructions de la boucle while se réalise.
Il est clair que la fonction T ne prend que des valeurs entières.
Ensuite, il est également évident que la fonction T est strictement décroissante, car d'une étape à l'autre, le compteur s est incrémenté d'une unité.
Enfin, on va montrer que la condition $T \leqslant 0$ marque la sortie de la boucle tant que.
En effet, à chaque lecture de la liste L, un plus grand élément de L est envoyé à la fin. À la première lecture, la nouvelle liste L présente un élément maximal à la fin.
On réitère ce raisonnement à la deuxième lecture, le raisonnement s'appliquant maintenant à la liste $L[:-1]$, un plus grand terme de L (le dernier en l'occurrence) n'étant plus déplacé par la suite.
Au bout de s étapes de lecture, les s derniers éléments de la liste L sont les s plus grands éléments, rangés dans l'ordre croissant.
Lorsque $T \leqslant 0$, alors $s \geqslant len(L)$ ce qui implique que tous les éléments de L sont correctement triés. La boucle tant que termine alors.

5.2.3 Exemple 3 : algorithme d'Euclide du pgcd

Rappelons l'algorithme d'Euclide pour le calcul de $a \wedge b$:

$\longrightarrow$ partir de deux entiers strictement positifs a et b

$\longrightarrow$ si $a > b$, remplacer a par $a - b$ et si $a < b$, remplacer b par $b - a$

$\longrightarrow$ faire l'étape précédente tant que $a \neq b$

$\longrightarrow$ lorsque a devient égal à b, on renvoie a.

On montre que cet algorithme termine en temps fini par exemple par la fonction de terminaison :

$$T = \max\{a, b\} \times (1 - \delta_{a,b}),$$

où $\delta_{a,b}$ est le symbole de Kronecker.
Il est clair que cette fonction ne prend que des valeurs entières puisqu'initialement, les quantités a et b sont entières et que d'une étape à l'autre, ces quantités restent entières.
Ensuite, en notant (a, b) et (A, B), les valeurs des variables avant et après la réalisation du bloc d'instructions, alors $a \neq b$, donc $\delta_{a,b} = 1$ et la fonction T vaut $\max\{a, b\}$ avant la réalisation du bloc.
Il s'agit de montrer que a et b sont strictement positifs. Si tel n'est pas le cas, on se place au premier moment où l'une des deux quantités a ou b est nulle. Par exemple, c'est la variable b qui devient nulle pour la première fois. Ainsi, $a > b = 0$. À l'étape précédente, on avait deux quantités notées α et β strictement positives avec :

$$b \neq \beta.$$

Nécessairement, c'est la variable b qui a été modifiée et donc :

$$a = \alpha \text{ et } b = 0 = \alpha - \beta,$$

contredisant le fait que la réalisation du bloc à l'étape précédente demandait la condition $\alpha \neq \beta$.

On en déduit que les variables a et b sont toujours strictement positives en début de réalisation du bloc d'instructions. On voit alors que la quantité $\max\{a, b\}$ diminue ou bien de b ou bien de a, donc la quantité T diminue strictement.

Enfin, si $T \leqslant 0$, alors cela impose $\delta_{a,b} = 1$, donc $a = b$, marquant la fin de la boucle tant que.

6 Les corrections partielles des algorithmes

6.1 Les invariants de boucle : définition

Considérons une boucle while du type :

```
while condition C :

    bloc instructions B
```

ou une boucle for du type :

```
for x in L :

    bloc d'instructions B
```

Un ***invariant de boucle*** est une assertion ou proposition logique $\mathscr{A}$ dépendant des paramètres mis en jeu dans le script et qui vérifie les conditions suivantes :

- avant la première réalisation du bloc d'instructions $\mathscr{B}$, l'assertion $\mathscr{A}$ est vraie
- à chaque réalisation complète du bloc d'instructions $\mathscr{B}$, l'assertion $\mathscr{A}$ reste vraie.

Un invariant de boucle $\mathscr{A}$ démontre que lorsque la boucle se termine, alors l'assertion $\mathscr{A}$ est vraie. Cela permet de démontrer que la sortie proposée par un algorithme correspond à ce que l'on voulait initialement.

6.2 Exemples d'application des variants de boucles

6.2.1 Exemple 1 : algorithme de division euclidienne

On considère la fonction suivante, applicable à des entiers naturels a et b et $b > 0$:

```
1  def DE(a,b) :
2      q=0
3      r=a
4      while r>=b :
5          q+=1
6          r-=b
7      return [q,r]
```

On voit que la fonction $T = r - b + 1$ est un variant de boucle.
Nous allons montrer qu'un variant de boucle est :

$$\mathscr{A} : \text{ « } a = bq + r \text{ et } \ r \geqslant 0. \text{ »}$$

En effet, avant l'amorce de la boucle while, on a bien $a = b \times 0 + a$ et $a \geqslant 0$.
Au cours du processus, en notant q et r les valeurs avant et Q et R les valeurs après la réalisation du bloc d'instructions, alors :

$$Q = q + 1 \text{ et } R = r - b.$$

On voit alors que :

$$a = bq + r = bQ + R.$$

De plus, comme on est dans la boucle while, alors $r \geqslant b$ donc $R \geqslant 0$.

La fonction de terminaison montre que cet algorithme termine en temps fini.
L'invariant de boucle montre qu'en sortie de la boucle while, l'assertion présente dans cet invariant est encore vraie.
Que devient cet invariant en sortie de boucle ?
La condition $r \geqslant b$ n'est plus réalisée, ce qui entraîne la condition $r < b$.
Enfin, on peut écrire :

$$a = bq + r \text{ et } r \geqslant 0,$$

donc $a = bq + r$ et $0 \leqslant r < b$.
Il s'agit bien de la division euclidienne de a par b.

6.2.2 Exemple 2 : algorithme du tri à bulles

La rédaction d'un variant de boucle pour la terminaison de ce tri suggère l'invariant de boucle suivant :
$\mathscr{A}$: « au cours de la $k^{ème}$ réalisation du bloc d'instructions, la liste L_k des k derniers éléments de la liste L est correctement triée et donne les k plus grands éléments de L ».
En effet, lorsque $k = 0$, la liste L_0 est vide donc correctement triée et on a ce qu'il faut.
Si à l'issue de la $k^{ème}$ étape, la liste L_k est correctement triée avec les plus grands éléments de L, à la lecture suivante, le plus grand terme de la liste des occurrences du début sera placé en fin de cette sous-liste et donc la sous-liste L_{k+1} sera correctement triée avec les $(k+1)$ plus grandes occurrences de la liste L.
On voit alors qu'au bout d'au maximum $len(L)$ réalisations de la boucle while, la liste $L_{len(L)} = L$ sera correctement triée.

6.2.3 Exemple 3 : algorithme d'Euclide du pgcd

On reprend l'algorithme traité ci-dessus pour le calcul de $a \wedge b$.
On note d le pgcd des entiers a et b.
On va montrer que l'assertion $\mathscr{A}$: « $a \wedge b = d$ » est un invariant de boucle.
En effet, avant l'amorce de la boucle tant que, l'assertion est vérifiée par définition de l'entier d.
Ensuite, d'une étape à l'autre le couple (a, b) est modifié en $(a - b, b)$ ou $(a, b - a)$ et un calcul d'arithmétique montre que :

$$d = (a - b) \wedge b = a \wedge (b - a).$$

En sortie de boucle tant que, on a $a = b$ et la valeur de sortie que l'on note δ vérifiera alors :

$$\delta \wedge \delta = d,$$

ce qui se réécrit :

$$\delta = d.$$

On a bien ce qu'il faut.

7 Exercices

7.1 Un exercice et sa solution complètement rédigée

7.1.1 Énoncé

On considère une liste L triée par ordre croissant et un flottant a.
On souhaite rédiger une fonction qui à partir de cette liste L et de ce flottant a renvoie un couple (i, j) d'indices tels que $0 \leqslant i < j < len(L)$ et $L[i] + L[j] = a$ lorsqu'un tel couple existe et renvoie le couple $(-1, -1)$ sinon.
On considère le script suivant, la liste L apparaissant dans la fonction étant supposée triée par ordre croissant.

```
def Recherche(L,a) :
    i,j=0,len(L)-1
    while i<j :
        if L[i]+L[j]==a :
            return [i,j]
        elif L[i]+L[j]<a :
            i+=1
        else :
            j-=1
    return [-1,-1]
```

1. Démontrer la terminaison de cette fonction.
2. Démontrer la correction de cette fonction.

7.1.2 Solution

1. Nous allons montrer que $T = j - i$ est un variant de boucle dans cette fonction.
   Initialement, la quantité $T = j - i$ vaut $len(L) - 1$ qui est entière.
   Au cours de l'algorithme, les variables (i, j) étant celles au début du bloc d'instructions et (i', j') étant celles à la fin du bloc d'instructions, alors soit $T = j - i$ est nul auquel cas, on sort de la boucle en renvoyant le résultat, soit $T > 0$ auquel cas que l'on ait $L[i] + L[j] > a$ ou $L[i] + L[j] < a$, on obtient quoi qu'il arrive :

   $$T' = j' - i' = T - 1.$$

   La quantité T est bien strictement décroissante.
   Enfin, la condition $T \leqslant 0$ marque la sortie de la boucle « tant que ». On a bien affaire à un variant de boucle qui prouve la terminaison.

2. Nous allons montrer que s'il existe un couple (k, ℓ) d'entiers différents entre 0 et $len(L)-1$ tels que $a = L[k]+L[\ell]$, alors la fonction ne peut pas renvoyer $[-1, -1]$.
Supposons l'existence d'un couple (k, ℓ) d'entiers différents entre 0 et $len(L) - 1$ tels que :
$$a = L[k] + L[\ell].$$
Parmi toutes les possibilités de tels couples (k, ℓ), on choisit un couple (k_0, ℓ_0) avec $k_0 < \ell_0$ tel que $a = L[k_0] + L[\ell_0]$ et avec l'entier k_0 minimal et l'entier ℓ_0 maximal pour cette propriété.
On montre déjà que l'assertion :
$$\mathscr{A} : \ « \, i \leqslant k_0 < \ell_0 \leqslant j \, »$$
est un invariant de boucle.
Le couple $(0, len(L) - 1)$ vérifie cette assertion.
Plaçons-nous à une étape de calcul, le couple d'entiers valant (i, j) au début du bloc d'instructions et ce couple valant ensuite (i', j').
On suppose l'assertion vérifiée pour le couple (i, j). On distingue deux cas, en supposant $a \neq L[i] + L[j]$:
- si $L[i] + L[j] < a$, alors $i' = i + 1$ et $j' = j$.
  Dans ce cas, il est impossible d'avoir $i' > k_0$ sinon, on aurait $i = k_0$ et :
  $$L[i] + L[j] < a = L[k_0] + L[\ell_0] = L[i] + L[\ell_0]$$
  et donc $L[j] < L[\ell_0]$ alors que la liste L est triée et $\ell_0 \leqslant j$.
  Ainsi, $i' \leqslant k_0 < \ell_0 \leqslant j'$ dans ce cas.
- si $L[i] + L[j] > a$, alors $i' = i$ et $j' = j - 1$.
  Dans ce cas, il est impossible d'avoir $j' < \ell_0$ sinon, on aurait $j = \ell_0$ et :
  $$L[i] + L[j] > a = L[k_0] + L[\ell_0] = L[k_0] + L[j]$$
  et donc $L[i] > L[k_0]$ alors que la liste L est triée et $i \leqslant k_0$.
  Ainsi, $i' \leqslant k_0 < \ell_0 \leqslant j'$ de nouveau dans ce cas.

Si le résultat de la fonction est $[-1, -1]$, il existe un moment juste avant la sortie de boucle où $j = i + 1$ puis i' devient $i + 1 = j$ ou bien j' devient $i - 1 = j$, pour avoir $i' = j'$ et sortir de la boucle.
Examinons cet instant juste avant la sortie.
$\longrightarrow$ Si i' devient $i + 1 = j$, cela implique $i \leqslant k_0 < \ell_0 \leqslant j$ et $L[i] + L[j] < a = L[k_0] + L[\ell_0]$, donc $i = k_0$, $\ell_0 = k_0 + 1$ et $j = \ell_0$. L'inégalité stricte $L[i] + L[j] < L[k_0] + L[\ell_0]$ n'a pas lieu.
$\longrightarrow$ Si j' devient $j - 1 = i$, cela implique $i \leqslant k_0 < \ell_0 \leqslant j$ et $L[i] + L[j] > a = L[k_0] + L[\ell_0]$, donc $i = k_0$, $\ell_0 = k_0 + 1$ et $j = \ell_0$ de nouveau. L'inégalité stricte $L[i] + L[j] > L[k_0] + L[\ell_0]$ n'a pas lieu non plus.
On vient de montrer que s'il existe un couple (i, j) d'entiers différents tel que $a = L[i] + L[j]$, alors la fonction ne peut renvoyer $[-1, -1]$.

La fonction renvoie donc un couple (i, j) tel que $i < j$ et $a = L[i] + L[j]$ et comme $i \leqslant k_0 < \ell_0 \leqslant j$, les entiers k_0 et ℓ_0 étant respectivement minimaux et maximaux, alors le couple d'entiers (i, j) renvoyé par la fonction est exactement le couple (k_0, ℓ_0).
Cette fonction teste s'il existe un couple d'entiers $(i < j)$ tel que :

$$a = L[i] + L[j]$$

et renvoie le couple (k_0, ℓ_0) convenable avec k_0 minimal et ℓ_0 maximal parmi tous les couples convenables, et renvoie $[-1, -1]$ si aucun couple (i, j) ne convient.

7.2 Les exercices à travailler

Exercice 1

On considère l'algorithme suivant :

- **entrée** : une liste L
- **algorithme** :

$Liste\_Temp \longleftarrow L$
$compteur \longleftarrow 0$
tant que $Liste\_Temp$ est non vide :
    enlever de $Liste\_Temp$ toutes les occurrences égales à $Liste\_Temp[0]$
    $compteur \longleftarrow compteur + 1$
fin tant que

- **sortie** : $compteur$

1. Déterminer un variant de boucle $T(\cdot)$ pour cet algorithme.
2. Montrer que l'assertion « $compteur$ est égal à la différence entre le nombre d'occurrences différentes dans la liste L et le nombre d'occurrences différentes dans la liste $Liste\_Temp$ » est un invariant de boucle.

Exercice 2

On considère la fonction définie par l'algorithme :

- **entrée** : un couple (a, b) avec a dans $\mathbb{N}$ et b dans $\mathbb{N}^*$
- **algorithme** :

$q \longleftarrow 0$
tant que $a - q \cdot b \geqslant b$:
    $q \longleftarrow q + 1$

fin tant que
- **sortie** : $a - q \cdot b$

Dans toute la suite, on notera $Reste(a, b)$, la sortie de cet algorithme en fonction de l'entrée (a, b).

1. Déterminer un variant de boucle pour cet algorithme $Reste(\cdot)$.
2. Montrer que l'assertion « $a - b \cdot q \in \mathbb{N}$ » est un invariant de boucle.
3. Que calcule alors l'algorithme $Reste(a, b)$?

On considère maintenant l'algorithme suivant :

- **entrée** : un entier $n \geqslant 2$
- **algorithme** :
  $compteur \longleftarrow 0$
  $n\_Temp \longleftarrow n$
  tant que $n\_Temp \neq 1$:
    $diviseur \longleftarrow 2$
    tant que $Reste(n\_Temp, diviseur) \neq 0$:
      $diviseur \longleftarrow diviseur + 1$
    fin tant que
    $n\_Temp \longleftarrow \dfrac{n\_Temp}{diviseur}$
    $compteur \longleftarrow compteur + 1$
  fin tant que
- **sortie** : $compteur$

4. (a) Montrer que $T = n\_Temp - 1$ est un variant de boucle pour la première boucle *tant que*.
   (b) En notant p_{n_Temp} le plus petit diviseur premier divisant n_Temp, montrer que $\widetilde{T} = p_{n\_Temp} - diviseur$ est un variant de la boucle *tant que* secondaire.
5. On note δ le nombre de diviseurs premiers comptés avec multiplicités de l'entier n.
   (a) Montrer que l'assertion :
   « $\delta - compteur$ est égal au nombre de diviseurs premiers de l'entier n_Temp »
   est un invariant pour la boucle *tant que* primaire.
   (b) Que calcule finalement cet algorithme ?

Exercice 3

Étant donnée une liste L, on appelle *groupe*, toute sous-liste maximale d'occurrences prises consécutivement dans L et égales. Par exemple, la liste $[1, 1, 2, 1, 2, 2, 2]$ contient quatre groupes, la première de longueur 2, les deux suivantes de longueur 1 et la dernière de longueur 3.

1. Rédiger une fonction qui à partir d'une liste L crée la liste des groupes de cette liste.
2. Rédiger une fonction qui à partir d'une liste crée la liste où les occurrences des groupes de longueur 1 sont rangées d'abord, puis les occurrences des groupes de longueur 2 (s'il y en a) sont placées ensuite, etc.
3. Faire des essais avec des listes aléatoires comportant 30 occurrences de nombres aléatoires dans $\{1, 2, 3\}$.