TP D'INFORMATIQUE 1
Syntaxe élémentaire de Python

1 Programmes et fonctions

Un **programme** est une suite d'instructions, qui va donc réaliser des calculs lorsqu'il sera exécuté, par exemple :

```
a = 84              #a prend la valeur 84
b = 52              #b prend la valeur 52
while a != b:       #tant que a est different de b
  if a > b:         #si a > b
    a = a - b       #a prend la valeur a - b
  else:             #sinon
    b = b - a       #b prend la valeur b - a
print(a)            #la valeur de a est affichee dans la console
```

L'**état de la mémoire** est la liste des valeurs actuellement affectées à chaque variable du programme. L'exécution d'un programme modifie l'état de la mémoire, ici avec des affectations (`=`).

1. Déterminer au papier l'évolution de l'état de la mémoire lors de l'exécution de ce programme.
2. Recopier et exécuter ce programme dans Pyzo et vérifier que l'affichage redonne la valeur déterminée sur papier.

Afin d'organiser le code et d'éviter d'avoir à le dupliquer, la plupart des programmes sont écrits dans des **fonctions**. Une fonction a un nom et des arguments (formels, c'est-à-dire juste des noms), sur le modèle suivant :

```
def pgcd(a, b):
  while a != b:
    if a > b:
      a = a - b
    else:
      b = b - a
  return a
```

Pour appeler la fonction, et donc l'exécuter, on lui donne des arguments effectifs, c'est-à-dire qui ont une valeur au moment de l'appel, sur le modèle suivant :

```
pgcd(84, 52)
```

On peut écrire un tel appel dans la console de Pyzo, mais aussi dans le code d'autres fonctions.

3. Écrire une fonction `distance` calculant la valeur absolue de la différence entre deux entiers (on utilisera un `if`). La tester avec un appel.
4. Écrire une fonction `distance_manhattan` prenant en argument quatre nombres a, b, c, d et renvoyant la somme de la distance entre a et c et de celle entre b et d (autrement dit, la distance Manhattan entre les points de coordonnées (a, b) et (c, d)). On utilisera des appels à la fonction précédente.

Le mot-clé `return` permet à une fonction de renvoyer un résultat, avec lequel on peut continuer un calcul lorsque cette fonction est appelée. Attention à ne pas le confondre avec le mot-clé `print`, qui ne fait que provoquer un affichage dans la console.

5. Vérifier que remplacer le `return` de la fonction `distance` par un `print` provoque une erreur lorsqu'on appelle `distance_manhattan`.

2 Conditions booléennes

Une **condition booléenne** a pour valeur `False` ou `True`. On en utilise notamment dans les structures conditionnelles (`if`) et les boucles conditionnelles (`while`). On forme ces conditions avec des comparaisons (`==`, `!=`, `<`, `<=`) et on les combine avec des opérateurs booléens (`and`, `or`). Attention à ne pas confondre `=` (affectation) et `==` (test d'égalité).

1. Remplacer les `?` dans la fonction suivante par l'opération pertinente, `=` ou `==` :

```
def mystere(a):
    b ? 2
    while a%b ? 0:      # % calcule le reste dans la division euclidienne
        b ? b + 1
    return b
```

2. Que calcule la fonction précédente ?
3. Écrire une fonction `minimum3` prenant en argument trois nombres, et renvoyant leur minimum. On n'utilisera pas la fonction `min` de Python.
4. Traduire en Python sous forme de condition booléenne les phrases suivantes :
   (a) l'entier n est pair.
   (b) Les nombres n et m sont de même signe.
   (c) Les points de coordonnées (x, y) et (z, t) ont même abscisse ou même ordonnée.
   (d) L'entier m est multiple de l'entier n (on vérifiera que le cas $n = 0$ ne provoque pas d'erreur).
   (e) Le point de coordonnées (x, y) appartient au disque de centre (z, t) et de rayon r.

3 Boucles

Une boucle permet d'exécuter plusieurs fois un même bloc de code (le **corps** de la boucle). Il existe deux types de boucles : conditionnelle (`while`) et inconditionnelle (`for`). En général, on écrit un `for` quand on peut savoir à l'avance combien de passages vont être faits dans la boucle, et un `while` dans le cas contraire. Voici un exemple de boucle `for` :

```
def factorielle(n):
  r = 1
  for i in range(2, n+1):       #pour i prenant les valeurs de 2 a n
    r = r * i
  return r
```

On notera bien que dans la syntaxe `for i in range(a, b)`, `i` prend les valeurs de `a` inclus à `b` **exclu**.

1. Déterminer l'évolution de l'état de la mémoire lors de l'appel `factorielle(5)`.
2. Écrire une fonction `somme_inverses_carres` prenant en argument un entier n et renvoyant la somme des inverses des carrés des n premiers entiers strictement positifs. On observera que cette somme converge vers $\pi^2/6 \approx 1.645$.

Pour $a \in \mathbb{N}^*$, on définit la suite de syracuse $(u_n)_{n\in\mathbb{N}}$ par :

$$u_0 = a, \quad \forall n \in \mathbb{N}, u_{n+1} = \begin{cases} u_n/2 & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{si } u_n \text{ est impair} \end{cases}$$

La conjecture de Syracuse énonce qu'une telle suite finit toujours par atteindre la valeur 1. Elle n'a jamais été mise en défaut, ni démontrée.

3. Écrire une fonction `nieme_terme` prenant en argument deux entiers a et n, et renvoyant le terme u_n de la suite de Syracuse commençant à a.
4. Écrire une fonction `temps_de_vol` prenant en argument l'entier de départ a, et renvoyant le premier entier n tel que $u_n = 1$. Vous devez obtenir un temps de vol de 111 pour $a = 27$.
5. Écrire une fonction `altitude_maximale` prenant en argument l'entier de départ a et renvoyant la valeur maximale prise par les termes de la suite. Vous devez obtenir 9232 pour $a = 27$.
6. Écrire une fonction `test_conjecture` prenant en argument un entier k, et renvoyant `True` après avoir vérifié que toutes les suites de Syracuse partant des entiers de 1 à k avaient fini par atteindre 1. Tester pour $k = 10^6$. On notera que tant que l'appel de la fonction n'a pas terminé, on ne sait pas si c'est parce que la conjecture est fausse, ou parce que le calcul nécessite plus de temps.
7. Écrire une fonction `test_conjecture_infini` ne prenant aucun argument, et testant la conjecture pour tout a dans $\mathbb{N}^*$. À chaque fois qu'une suite finit par atteindre 1, la fonction provoquera un affichage dans la console indiquant que cette valeur de a a validé la conjecture, ainsi que le temps de vol associé. Par construction, cette fonction ne termine jamais, on pourra interrompre son exécution avec le raccourci `Ctrl + i`.
8. Écrire une fonction `somme_chiffres` prenant en argument un entier n, et renvoyant la somme des chiffres de son écriture décimale. Par exemple, `somme_chiffres(5391)` doit renvoyer 18 ($= 5 + 3 + 9 + 1$). On pourra utiliser des divisions euclidiennes par 10.
9. Écrire une fonction `est_un_nombre_palindrome` prenant en argument un entier n, et testant (c'est-à-àdire renvoyant `True` ou `False` si n est un nombre palindrome, c'est-à-dire si les chiffres dans l'écriture décimale de n sont les mêmes en lisant de gauche à droite et de droite à gauche.
   Ainsi, cette fonction doit renvoyer `True` sur 12521 ou sur 3443, et `False` sur 1271