Consigne : vous devez rendre une copie double avec vos réponses aux questions et tous les codes demandés écrits à la main en respectant scrupuleusement l'indentation.

Pour tester vos programmes :

- *Installez la distribution Anaconda depuis* https://www.anaconda.com/distribution/ *puis l'environnement de programmation Pyzo depuis* https://pyzo.org/start.html
- *Si vous ne parvenez pas à installer Anaconda, installez la distribution standard depuis* https://www.python.org/downloads/ *et utilisez l'environnement de développement par défaut qui s'appelle Idle.*
- *Sinon utilisez l'interpréteur en ligne* https://repl.it/languages/python3

1 Boucles inconditionnelles `for`

Méthode

La traduction en Python de la **boucle Pour** :

```
Pour k allant de 1 a 10 repeter
    Bloc d'instructions
FinPour
```

est :

```
# flux parent
for k in range(1, 11):
    # bloc d'instructions
# retour au flux parent
```

Le bloc d'instructions de la boucle est délimité par le symbole : et par son niveau d'indentation.

On remarque l'utilisation `range(1, 11)` alors qu'on attendrait `range(1, 10)`.

En fait `range(1, 11)` va parcourir avec un incrément de 1 tous les entiers n tels que $1 \leqslant n < 11$.

Il faut bien se souvenir que la borne supérieure de `range(n, m)` est exclue mais il est facile de retenir que le nombre de tours de boucles est $m - n$.

Par exemple, pour calculer la somme des tous les entiers consécutifs de 100 à 200, on peut écrire :

```
somme = 0
for k in range(100, 201):
    somme = somme + k
print("La somme est ", somme)
```

S'il s'agit juste de répéter `n` fois un bloc d'instructions on utilise le raccourci `range(n)` au lieu de `range(0, n)` ou de `range(1, n + 1)`. Par exemple pour dire 100 fois "Merci!", on peut écrire :

```
for k in range(100):
    print("Merci !")
print("Ouf!")
```

La fonction `range` offre un troisième paramètre optionnel qui permet de changer l'incrément par défaut qui est 1.

Par exemple, pour calculer la somme des tous les entiers pairs consécutifs entre 100 et 200, on peut écrire :

```
sommePair = 0
for k in range(100, 201, 2):
    sommePair = sommePair + k
print("La somme est ", sommePair)
```

Au passage, on a vu un exemple de bloc d'instruction de boucle qui utilise la variable de boucle.

On peut même utiliser un incrément négatif, par exemple pour énumérer les entiers tels que $10 \geqslant n \geqslant 0$ avec un pas de -1 :

```
print("Je sais compter à l'envers")
for k in range(10, 0, -1):
    print(k)
print("J'ai décompté de 10 inclus à 0 exclu.")
```

 Une bonne pratique de programmation est de ne jamais modifier la variable de boucle dans le bloc de boucle, même si avec range c'est transparent car celui-ci reprend la main à chaque tour pour affecter à la variable de boucle la valeur attendue.

Méthode

Un entier naturel est divisible par 7 si son reste dans la division euclidienne par 7 est nul. L'opérateur % permet en `Python` de déterminer le reste de la division euclidienne entre deux entiers naturels.

```
In [5]: 14 % 7
Out[5]: 0

In [6]: 16 % 7
Out[6]: 2
```

L'opérateur de comparaison == permet alors de tester si un entier naturel est divisible par 7, la valeur logique retournée est `False` ou `True`, elle est de type `booléen`.

```
In [7]: 14 % 7 == 0
Out[7]: True

In [8]: 16 % 7 == 0
Out[8]: False
```

On peut combiner des tests avec des opérateurs logiques comme `not`, `or` ou `and`.

```
In [9]: 14 % 7 == 0 and 14 % 5 == 0
Out[9]: False

In [10]: 14 % 7 == 0 or 14 % 5 == 0
Out[10]: True

In [11]: not 14 % 7 == 0
Out[11]: False
```

Ces opérateurs logiques sont caractérisés par leur table de vérité :

a	b	a and b
True	True	True
True	False	False
False	True	False
False	False	False

a	b	a or b
True	True	True
True	False	True
False	True	True
False	False	False

a	not a
True	False
False	True

Exercice 1

1. Écrire un programme Python qui affiche les entiers pairs compris entre 0 et 100 inclus dans l'ordre croissant.
2. Écrire un programme Python qui affiche les entiers pairs compris entre 100 et 0 inclus dans l'ordre décroissant.
3. Recopier et compléter le programme Python ci-dessous pour que la variable s contienne en sortie de boucle la somme des puissances 4 des entiers successifs entre 1 et 100 : $\sum_{k=1}^{100} k^4 = 1 + 2^4 + 3^4 + \ldots + 100^4$.

```
s = ....
for n in range(.............):
    s = ....................
```

4. Si on liste tous les entiers naturels inférieurs à 20 qui sont multiples de 5 ou de 7, on obtient 5, 7, 10, 14 et 15. La somme de ces nombres est 51. Écrire un programme Python qui détermine la somme de tous les multiples de 5 ou de 7 inférieurs à 42455.

2 Fonctions

Méthode

Lorsqu'on a besoin de réutiliser tout un bloc d'instructions, on peut l'encapsuler dans une **fonction**. On étend ainsi le langage avec une nouvelle instruction. Une fonction sert à factoriser du code, à rendre le programme plus lisible et plus facile à maintenir et à partager. C'est un outil de modularité.

Pour déclarer une fonction, on définit son **en-tête** avec son **nom** et des **paramètres formels** d'entrée. Viennent ensuite le bloc d'instructions et une valeur de retour. En Python on définit une fonction ainsi :

```
def mafonction(parametre1, parametre2):
    #bloc d'instructions (optionnel)
    return valeur
```

Par exemple une fonction carre qui prend en paramètre un nombre x et qui retourne son carré, s'écrira :

```
def carre(x):
    return x ** 2
```

Une fonction peut prendre plusieurs paramètres. Par exemple une fonction carre_distance_origine(x,y qui prend en paramètres deux nombres x et y et qui retourne le carré de la distance d'un point de coor-

données `(x, y)` à l'origine d'un repère orthonormal, s'écrira :

```
def carre_distance_origine(x, y):
    return x ** 2 + y ** 2
```

On a parfois besoin d'utiliser des fonctions de Python qui ne sont pas chargées pas défaut. Ces fonctions sont stockées dans des programmes Python appelées **modules** ou **bibliothèques**. On les importe dans le programme avec la directive `import` Par exemple le module `math` contient les fonctions mathématiques usuelles .

```
#import de la fonction sqrt du module math
from math import sqrt
racine = sqrt(2)
#Pour importer toutes les fonctions de math, on écrit from math import *
```

Deuxième façon

Exercice 2

1. Parmi les fonctions ci-dessous déterminer celles qui retournent le terme de rang `n` de la suite définie par

$$\begin{cases} u_0 = 734 \\ \forall n \in \mathbb{N}, \quad u_{n+1} = \dfrac{2}{3}u_n + 100 \end{cases}$$

```
def suiteU1(n):
    u = 734
    for k in range(n):
        u = 2 / 3 * u + 100
        return u
```

```
def suiteU2(n):
    u = 734
    for k in range(n):
        u = 2 / 3 * u + 100
    return u
```

```
def suiteU3(n):
    u = 734
    for k in range(1, n + 1):
        u = 2 / 3 * u
        u = u + 100
    return u
```

```
def suiteU4(n):
    u = 734
    for k in range(1, n + 1):
        u = u + 100
        u = 2 / 3 * u
    return u
```

2. Écrire une fonction `suiteV(n)` pour qu'elle retourne le terme de rang `n` de la suite définie par

$$\begin{cases} v_0 = 10 \\ \forall n \in \mathbb{N}, \quad v_{n+1} = \dfrac{2}{3}v_n + n + 1 \end{cases}$$

3 Boucle conditionnelles `while`

Méthode *Boucle `Tant Que` en Python*

La traduction de la boucle `Tant Que` en Python est :

```
# flux parent
while condition:
    # bloc d'instructions avec indentation
# retour au flux parent
```

Le bloc d'instructions de la boucle est délimité par le symbole : et par son niveau d'indentation.

Par exemple, pour déterminer le plus petit entier $n \geqslant$ tel que $n^3 > n^2 + 1000n$ (*condition d'arrêt*), on écrit une boucle Tant Que dont la *condition d'entrée de boucle* est la *négation de la condition d'arrêt* :

```
n = 0
while n ** 3 <= n ** 2 + 1000 * n:    # test d'entree de boucle
    n = n + 1
```

 On peut aboutir à une **boucle infinie** si la condition d'entrée de boucle n'est jamais vérifiée.

Exercice 3

1. On considère la fonction `mystere(n)` ci-dessous :

```
def mystere(n):
    s = 0
    while n > 0:
        s = s + n % 10
        n = n // 10
    return s
```

a. Décrire dans un tableau l'évolution des variables `s` et `n` au cours de l'exécution de `mystere(734)`. Que représente la valeur retournée par la fonction `mystere` ?

b. En s'inspirant de la fonction précédente, écrire une fonction `miroir(n)` qui retourne le miroir de l'entier naturel `n` passé en paramètre c'est-à-dire le nombre `n` écrit de droite à gauche. Par exemple, `miroir(734) = 437`.

c. Écrire un programme qui détermine le grand grand entier naturel `n` inférieur à 10 millions ayant la propriété : `miroir(n) = 4 × n`.

2. Chaque nouveau terme de la suite de Fibonacci est généré en ajoutant les deux termes précédents. En commençant avec 1 et 1, les 10 premiers termes sont les suivants : 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

a. Écrire une fonction `fibo(n)` qui retourne le terme de rang `n` de la suite de Fibonacci.

b. Écrire une fonction `sommeImpairFibo(m)` qui retourne la somme des termes impairs de la suite de Fibonacci dont la valeur ne dépasse pas `m`.

```
In [10]: sommeImpairFibo(4000000)
Out[10]: 4613732
```