1 Structures de contrôle

L'instruction `if` permet de soumettre l'exécution d'une instruction ou d'un bloc d'instructions à une condition.

```
>>> solde_compte = -100
    if solde_compte < 0:
        print("Vous êtes à découvert.")
        print("Veuillez passer à la banque.")
    print("Bonne journée.")

 Vous êtes à découvert.
 Veuillez passer à la banque.
 Bonne journée.
```

Le bloc d'instructions soumis à condition est délimité par l'*indentation*. Par rapport à l'instruction `if`, on décale d'un même nombre d'espaces chaque instruction faisant partie du bloc d'instructions. Par convention, nous choisissons une indentation de 4 espaces. L'instruction suivant la fin du bloc doit avoir le même niveau d'indentation que l'instruction `if`. Le programme précédent vous souhaitera donc une bonne journée quelque soit l'état de votre compte.

Il est possible d'exécuter un bloc d'instructions dans le cas où la condition n'est pas vérifiée.

```
>>> est_homme = False
    if est_homme:
        salutation = "Monsieur"
    else:
        salutation = "Madame"
    print("Bonjour " + salutation + ".")

 Bonjour Madame.
```

Enfin, il est possible d'exécuter différents blocs selon plusieurs conditions.

```
>>> note = 13
    if note >= 16:
        print("Mention Très Bien.")
    elif note >= 14:
        print("Mention Assez Bien.")
    elif note >= 12:
        print("Mention Bien.")
    elif note >= 10:
        print("Vous avez votre Bac.")
    else:
        print("Same player shoot again!")

 Mention Assez Bien
```

Seul le bloc correspondant à la première condition vérifiée est exécuté.

2 Boucles inconditionnelles

Il est possible de répéter plusieurs fois la même séquence d'instructions en utilisant une boucle `for`. Pour délimiter le groupe d'instructions qui seront répétées, on indente ces instructions d'un même nombre d'espaces. Il est courant d'utiliser 2 espaces. La première instruction ne faisant pas partie de la boucle doit utiliser le même niveau d'indentation que la ligne `for`. Dans le film « The Shining » de Stanley Kubrick, adapté du roman de Stephen King, le personnage principal succombe petit à petit à la folie :

```
>>> for k in range(2):
        print("All work and no play")
        print("makes Jack a dull boy.")
    print("Jack Torrance")

 All work and no play
 makes Jack a dull boy.
 All work and no play
 makes Jack a dull boy.
 Jack Torrance
```

Pour calculer le n-ième terme de la suite définie par

$$u_0 = 1 \qquad \text{et} \qquad \forall n \in \mathbb{N} \quad u_{n+1} = \cos(u_n)$$

on utilisera le programme suivant.

```
>>> import numpy as np
>>> u = 1.0
    n = 1000
    for k in range(n):
        u = np.cos(u)
    print("u_" + str(n) + " =", u)
```

Il est souvent utile d'avoir un index prenant les valeurs entières successives lors de la boucle. Ainsi, pour afficher les premiers termes de la suite définie par

$$\forall n \in \mathbb{N} \quad u_n = \sum_{k=0}^{n} \frac{1}{k!}$$

on utilise le programme suivant.

```
>>> n = 10
    u = 0.0
    for k in range(n):
        u = u + 1 / math.factorial(k)
        print("u_" + str(k + 1) + " =", u)
```

La variable `k` va prendre successivement les valeurs $0, 1, 2, \ldots, n-1$. On peut faire commencer la variable `k` à une valeur autre que 0 en utilisant `range(a, b)`. Dans ce cas, la variable `k` va successivement prendre les valeurs $a, a+1, \ldots, b-1$. Ainsi, pour calculer le n-ième terme de la suite

$$\forall n \in \mathbb{N} \quad u_n = \sum_{k=1}^{n} \frac{1}{k^2}$$

on utilisera le programme

```
>>> somme = 0.0
    n = 1000
    for k in range(1, n + 1):
        somme = somme + 1 / k**2
>>> somme
```

Plus généralement, si $\delta > 0$, l'instruction `range(a, b, delta)` est utilisée pour boucler sur les entiers $a, a+\delta, a+2\delta$ jusqu'au plus grand entier de la forme $a + k\delta$ strictement inférieur à b. Lorsque $\delta < 0$, cette même instruction permet d'itérer sur les entiers $a, a+\delta, a+2\delta$ jusqu'au plus petit entier de la forme $a + k\delta$ strictement supérieur à b. En particulier, le programme suivant permet de calculer la même somme en sommant dans l'ordre inverse.

```
>>> somme = 0.0
    n = 1000
    for k in range(n, 0, -1):
        somme = somme + 1 / k**2
>>> somme
```

Remarquons que le résultat est légèrement différent du premier. Ce phénomène est dû à la non associativité de l'addition sur les nombres flottants, conséquence des arrondis effectués à chaque opération. Vous lisez bien, j'ai bien parlé de la non associativité de l'addition sur les flottants et non de sa non commutativité. Et pour cause, l'addition sur les nombres flottants est bien commutative.

Exercice 1

1. Écrire une boucle permettant de calculer $n!$.
2. Écrire une boucle permettant de calculer le produit de tous les nombres impairs entre 0 et 100.

Exercice 2

1. Écrire une boucle qui calcule la valeur de

$$B_n = \sum_{k=0}^{n} \frac{(-1)^k}{k!}$$

2. Écrire une boucle qui calcule B_n et utilise moins de n multiplications.

Il est possible d'imbriquer les boucles les unes dans les autres. Vous pouvez par exemple générer les tables de multiplication très facilement de la manière suivante.

```
>>> for a in range(1, 10):
        for b in range(1, 10):
            print(a, "*", b, "=", a * b)
```

Exercice 3

Complétez ce programme pour que les entiers qu'il affiche soient dans l'ordre croissant.

```
for _ in range(10):
    for _ in range(10):
        for _ in range(10):
            print(a + 10 * b + 100 * c)
```

3 Boucles conditionnelles

Les boucles `for` nous ont permis d'exéctuer plusieurs fois un bloc d'instructions lorsque le nombre d'itérations est connu avant de rentrer dans la boucle. Lorsque ce nombre n'est pas connu avant de rentrer dans la boucle, typiquement lorsque l'on doit exécuter un bloc tant qu'une condition est vérifiée, on utilise l'instruction `while`. Le programme suivant calcule le nombre de chiffres utilisés pour écrire l'entier u en base 10.

```
>>> u = 123
    m = 0
    v = 1
    while v <= u:
        v = v * 10
        m = m + 1
>>> m
```

La condition « `v <= u` » est évaluée en début de boucle. Si elle est vérifiée, le bloc d'instruction est exécuté. Puis la condition est à nouveau évaluée. La boucle s'arrête lorsque la condition est fausse. Pour la valeur initiale de 123, la variable `v` prend successivement les valeurs 1, 10, 100 et 1000. On passe donc 3 fois dans la boucle ce qui nous assure que le nombre s'écrit bien avec 3 chiffres.

Exercice 4

1. Écrire une boucle de calcul du quotient et du reste de la Division euclidienne de $m = 15466$ par $p = 243$, par l'algorithme des différences successives.
2. On suppose que `n` et `p` sont des variables entières (naturelles et non nulles) préalablement définies. Écrire une boucle qui détermine, la plus grande puissance de p qui divise n.

Contrairement à ce qui se passe pour une boucle conditionnelle pour laquelle on est assuré de sortir de la boucle, il est tout à fait possible qu'une boucle inconditionnelle ne sorte jamais de la boucle. On dit alors que le programme part en boucle infinie. Dans le film, « Groundhog day », le personnage joué par Bill Muray se réveille chaque matin au son de la radio qui lui annonce le jour de la marmotte.

```
>>> while True:
        print("This is Groundhog day!")
```

Vous pourrez interrompre le programme en appuyant à la fois sur la touche « control » et la touche « c ».

Le calcul du pgcd par l'algorithme d'Euclide se fait en remplaçant `a` et `b` respectivement par `b` et `a % b`. Lorsque `b` est nul, on obtient le pgcd dans la variable `a`.

```
>>> a = 42
    b = 18
    while b != 0:
        c = a % b
        a = b
        b = c
>>> a
```

Exercice 5

On note pour tout $n \in \mathbb{N}^*$:

$$S_n = \sum_{k=1}^{n} \frac{1}{k} \text{ et } u_n = S_n - \ln(n) \text{ et } v_n = u_n - \frac{1}{n}$$

On admet que (u_n) et (v_n) tendent vers la même limite γ et que l'on a

$$\forall n \in \mathbb{N}^* \quad v_n \leqslant \gamma \leqslant u_n.$$

Écrire une boucle qui calcule une approximation à ε près de γ.

La suite de Syracuse est définie en initialisant u_0 avec un entier m et en définissant :

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

L'expérience montre que, quel que soit l'entier m choisit, la suite fini par prendre les valeurs 1, 4, 2, 1, 4, 2, 1, etc. Afin de vérifier ce phénomène, on écrit un programme permettant d'afficher toutes les valeurs de la suite, jusqu'à ce que $u_n = 1$.

```
>>> u = 17
    n = 0
    while u != 1:
        if u % 2 == 0:
            u = u // 2
        else:
            u = 3 * u + 1
        n = n + 1
        print("u_" + str(n) + " =", u)
```

À ce jour, personne n'a pu montrer que la suite de Syracuse finissait par prendre la valeur 1 quelle que soit la première valeur choisie. On a cependant vérifié que c'était bien le cas pour de nombreux entiers. Ce programme est un exemple très simple pour lequel aucune preuve de terminaison n'existe à ce jour.

Voici enfin un exercice pour vérifier que ce chapitre est bien compris.

Exercice 6

Que font les programmes suivants ?

```
res = 0
n = 30
for i in range(n):
    res = res + i
print(res)
```

```
a = 3
res = a
n = 30
for i range(1, n + 1):
    res = res * a
print(res)
```

```
a = 3
res = a
n = 10
for i in range(n + 1):
    res = res + a
    print(res)
```

```
res = 1
a = 13
n = 50
for i in range(1, n + 1):
    res = res * a
    print(res)
```

```python
res = 1
a = 13
n = 50
for i in range(2, n + 1, 2):
    res = res * a
```

```python
n = 30
res = 0
for i in range(1, n + 1):
    if i % 2 == 0:
        res = res + i
    else:
        res = res - i
print(res)
```

```python
res = 0
i = 0
a = 4
n = 100
while i <= n:
    res = res + i
    i = i + 1
print(res)
```