TP 4
Boucles for

Dans ce TP nous approfondissons sur les boucles, et les applications mathématiques aux calculs de suites et de sommes.

À partir de maintenant, dans les exercices, les programmes sont la plupart du temps contenus dans des **fonctions** avec éventuellement des paramètres et une valeur de retour. Cela permet de les ré-utiliser autant qu'on veut en faisant varier un paramètre, mais sans utiliser `input()`.

I Boucles simples

La boucle `for` sert à répéter un bloc d'instructions un certain nombre de fois. Dans sa syntaxe de base, elle prend la forme

```
for i in range(a, b):
    instructions
```

qui est exactement équivalente à

```
i = a
while i < b:
    instructions
    i = i + 1
```

Autrement dit elle répète en faisant varier l'indice i avec $a \leqslant i < b$. Nous verrons qu'elle est un peu plus pratique quand on sait à l'avance combien de fois on veut répéter les instructions. Essai :

```
for i in range(1, 5):
    print(i)
```

produit bien

```
1
2
3
4
```

Un fait remarquable auquel il faudra s'habituer est que **la borne `b` est exclue**. Il existe aussi `range(n)` qui est exactement équivalent à `range(0, n)`.

À retenir

`range(n)` répète n fois, avec `i` qui varie entre 0 et $n-1$ inclus.

Éventuellement, un troisième argument `range(a, b, r)` consiste à faire des sauts de taille `r` au lieu de 1. Cela correspond, dans la traduction en terme de boucle `while` ci-dessus, à remplacer la ligne `i = i + 1` par `i = i + r`. On ne l'utilisera pas tous les jours.

Éventuellement aussi, le caractère `_` permet de ne pas donner de nom à la variable indice de la boucle. Ainsi `for _ in range(n)` a réellement le sens de « répéter `n` fois ».

C'est l'heure de démarrer le TP !

```
nom = "..." # insérez votre prénom
for _ in range(3):
      print("Au travail", nom)
```

Exercice 1
Vous connaissez la chanson

```
1 kilomètre à pieds, ça use, ça use,
1 kilomètre à pieds, ça use les souliers.
...
```

Écrire une fonction `kilomètres(n)` qui affiche les paroles pendant `n` kilomètres.

Une application plus sérieuse des boucles `for` est pour le calcul des suites et des sommes, que nous avons déjà abordé. L'exemple de base est la fonction suivante qui calcule et renvoie le nombre 2^n :

```
def puissance2(n):
    u = 1
    for i in range(n):
        u = 2 * u
    return u
```

par exemple

```
>>> puissance2(5):
32
```

Il faut remarquer que :

- La variable `u` représente la valeur de 2^i en entrant dans la boucle, en particulier c'est 2^0 avant de rentrer dans la boucle.
- La variable `u` représente 2^{i+1} à la fin de la boucle. L'étape `u = 2 * u` fait « avancer d'un cran » la suite.
- Lors de la dernière itération, $i = n - 1$ et donc à la fin `u` représente bien 2^n.
- … d'ailleurs on répète n fois l'opération de multiplier `u` par 2, en partant de 1, donc le résultat est bien 2^n.

Avertissement

Il est **très important** de se poser ce genre de questions quand on utilise des boucles pour calculer des suites et des sommes, notamment ce qui se passe au début et à la fin. Cela sera source de **nombreux** problèmes et pièges. Une possibilité est d'écrire clairement, en commentaire ou sur son brouillon, une phrase telle que « `u` représente 2^i en début de boucle ».

Les exercices suivants sont les plus proches possible des questions posées à l'écrit. Pour tester son programme, on pourra aussi afficher les valeurs de la variable `u` dans la boucle.

Exercice 2
Écrire une fonction `suite(n)` qui renvoie le terme u_n de la suite définie par $u_0 = 2$ et $\forall n \in \mathbb{N}$, $u_{n+1} = 1 + \frac{1}{u_n}$.

Exercice 3
La **factorielle** de l'entier $n \geqslant 1$, notée $n!$, est le produit $1 \times 2 \times \cdots \times n$. On convient que pour $n = 0$ c'est 1 (c'est cohérent).

Écrire une fonction `factoriel(n)` qui calcule la factorielle de l'entier n.

Exercice 4
La **suite de Fibonacci** est la suite $(F_n)_{n\in\mathbb{N}}$ définie par $F_0 = 0$, $F_1 = 1$ et la relation de récurrence

$$\forall n \in \mathbb{N}, \quad F_{n+2} = F_{n+1} + F_n$$

Pour calculer les termes d'une telle suite on a besoin d'une boucle `for` avec *deux* variables : `u` qui représente la valeur de F_i et `v` qui représente F_{i+1}.

Écrire une fonction `fibonacci(n)` qui calcule le terme F_n.

Dans le cas des sommes, la variable qui représente le terme de la suite se nomme plutôt `S` et s'appelle **variable accumulatrice**, car elle accumule la somme de plus en plus de termes. Elle est toujours initialisée à 0 (si aucune somme ne se produit, la valeur de retour doit être 0) et dans la boucle elle évolue selon une formule du type `S = S + ...` (le nouveau terme à sommer).

Exercice 5
Écrire une fonction `somme_cubes(n)` qui calcule et renvoie le résultat de la somme $1^3 + 2^3 + \cdots + n^3$.

Bonus : vérifier (avec une boucle) sur les 10 premières valeurs de n que la somme est bien égale à $\frac{n^2(n+1)^2}{4}$.

Exercice 6
Écrire une fonction `somme_inverse_factoriel(n)` qui calcule et renvoie le résultat de la somme des $\frac{1}{k!}$ pour $0 \leqslant k \leqslant n$, c'est à dire $1 + 1 + \frac{1}{2} + \frac{1}{6} + \cdots + \frac{1}{n!}$ (avec $\frac{1}{0!} = 1$ et $\frac{1}{1!} = 1$). Tester la fonction pour des valeurs de plus en plus grandes.

Bonus : pouvez-vous l'écrire *sans* faire appel à la fonction `factoriel()` précédente ?

Remarque. Parfois, on veut calculer les termes de la suite et arrêter la boucle quand une certaine condition est vérifiée, par exemple quand u_n dépasse une certaine valeur fixée à l'avance. Dans ce cas il faut revenir à une boucle `while` — la partie qui calcule les termes de la suite reste la même, et on rajoute à la main l'initialisation de `i` avant la boucle et l'incrémentation `i = i + 1` dedans (pour être cohérent, l'incrémentation *en fin de boucle*). Revoir pour cela les TP précédents et leur correction.

II Boucles doubles

Pour parcourir l'ensemble de toutes les valeurs possibles pour deux indices indépendants `i` et `j`, il est nécessaire d'utiliser une **boucle double**, ce qui n'est rien de plus qu'une boucle située dans le corps d'une autre boucle. Mais il est important d'en comprendre précisément le mécanisme.

Exercice 7
Tester et comparer les deux programmes suivants :

```
for i in range(3):
    for j in range(3):
        print("i =", i, "j =", j)
```

```
for j in range(3):
    for i in range(3):
        print("i =", i, "j =", j)
```

Que se passe-t-il ?

Observer que ce sont deux ordres naturels pour énumérer le carré d'indices $(i, j) \in \{0, 1, 2\} \times \{0, 1, 2\}$.

$i \setminus j$	0	1	2
0	`i = 0 j = 0`	`i = 0 j = 1`	`i = 0 j = 2`
1	`i = 1 j = 0`	`i = 1 j = 1`	`i = 1 j = 2`
2	`i = 2 j = 0`	`i = 2 j = 1`	`i = 2 j = 2`

Exercice 8

On souhaite écrire une fonction (sans arguments ni valeur de retour) qui affiche la table de multiplication, la plus classique, des nombres de 1 à 10.

1. Écrire une fonction `table()` qui affiche la liste de tous les produits $i \times j$ pour i et j entre 1 et 10, produisant par exemple le résultat suivant :

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
...
1 * 10 = 10
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
...
10 * 10 = 100
```

2. On souhaite maintenant formater la table en carré, dans le but de produire le résultat suivant :

```
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

   Pour cela on a besoin de savoir que
   - Dans la commande `print()`, rajouter l'argument final `end=""` désactive le saut de ligne automatique (en fait, cela remplace le caractère de saut de ligne ajouté automatiquement par la chaine vide, donc il ne se passe rien) : `print(x, end="")`.
   - Et `print()` sans argument effectue un simple saut de ligne.

   Écrire cette fonction, qu'on appellera `table_carré()`.

3. Bonus : on souhaite enfin aligner correctement les colonnes, pour obtenir

```
  1   2   3   4   5   6   7   8   9  10
  2   4   6   8  10  12  14  16  18  20
  3   6   9  12  15  18  21  24  27  30
  4   8  12  16  20  24  28  32  36  40
  5  10  15  20  25  30  35  40  45  50
  6  12  18  24  30  36  42  48  54  60
  7  14  21  28  35  42  49  56  63  70
  8  16  24  32  40  48  56  64  72  80
  9  18  27  36  45  54  63  72  81  90
 10  20  30  40  50  60  70  80  90 100
```

   Pour cela on utilise les chaines formatées : la syntaxe `print(f"{x:4d}")` affiche le nombre entier `x` en prenant l'espace de 4 caractères, serré à droite. Si `x = 10` la chaine `f"{x:4d}"` est ainsi remplacée par `"␣␣10"` où le symbole ␣ indique un espace (le `f` devant signifie que la chaine doit être formatée, le `x` est le nom de la variable, et `4d` est l'un des nombreux codes de formatage signifiant ici *entier affiché sur 4 caractères*).

   Écrire cette fonction, qu'on appellera `table_carré_joli()`.

Exercice 9
Un **triplet pythagoricien** est la donnée de trois nombres $(a, b, c) \in \mathbb{Z}^3$ tels que

$$a^2 + b^2 = c^2$$

On souhaite afficher tous les triplets pythagoriciens. Remarquons que :
- On peut toujours changer l'un des nombres en son opposé et on obtient toujours un triplet pythagoricien. On se limitera donc aux triplets de nombres positifs.
- Il y a des solutions évidentes pour $a = 0$ (avec $b = c$) et pour $b = 0$ (avec $a = c$). On peut donc même chercher les solutions où a et b sont strictement positifs.
- Il peut y avoir une infinité de solutions. Il faut donc fixer un entier N et chercher les solutions (a, b, c) avec $a \leqslant N$, $b \leqslant N$, $c \leqslant N$.

Écrire une fonction `pythagore(N)` qui affiche tous les triplets pythagoriciens avec (a, b, c) comme ci-dessus.

Bonus : chaque triplet apparait deux fois, en échangeant a et b. Pouvez-vous les énumérer de telle façon à ce que chaque triplet n'apparaisse qu'une seule fois ?

III D'autres types d'itération

La boucle `for` fait bien plus que tout cela…

Un **tuple** est la donnée de deux ou plusieurs objets Python, entre parenthèses et séparés par des virgules. Cela forme un nouveau type qui correspond directement à la notion de produit cartésien en mathématiques. Par exemple la variable

```
>>> t = (4, 17)
>>> type(t)
<class 'tuple'>
```

est construite à partir de deux nombres et est de type `tuple`. Il est alors possible de la **récupérer ses valeurs** avec la syntaxe

```
>>> (x, y) = t
>>> x
4
>>> y
17
```

qui va alors créer d'un coup deux nouvelles variables `x` et `y`. Attention, ce n'est pas du tout la même chose que `t = (x, y)`, qui elle crée un tuple à partir de deux variables existantes `x` et `y`. Les composantes d'un tuple peuvent en fait être de tous les types vus précédemment.

Il possible d'**itérer sur les éléments d'un tuple**. Dans le boucle suivante, la variable `x` prend successivement les valeurs des composantes de `t`. Testez-la !

```
t = ("mathématiques", "physique-chimie", "SVT"):
for x in t:
    print("J'aime le cours de", x, "car je suis en BCPST.")
```

Exercice 10
Écrire un programme, le plus court possible, qui affiche les valeurs de `somme_inverse_factoriel(n)` de § I pour $n = 1$, $n = 2$, $n = 3$, $n = 5$, $n = 10$ et $n = 20$.

Un autre intérêt des tuples est de fournir une syntaxe rapide et pratique pour échanger deux variables : c'est tout simplement

```
(y, x) = (x, y)
```

sans avoir besoin d'écrire une variable intermédiaire.

Ils sont aussi faciles à utiliser pour une fonction qui doit renvoyer *deux* valeurs.

Exercice 11
On reprend la suite de Fibonacci de l'exercice 4. Écrire une fonction `quotients(n)` qui renvoie le couple $\left(\frac{F_{n+1}}{F_n}, \frac{F_{n+2}}{F_{n+1}}\right)$. Puis tester cette fonction pour tous les n entre 1 et 10. Qu'observe-t-on ?

Enfin nous verrons au TP suivant qu'une boucle `for` est utile pour obtenir uns par uns les caractères d'une chaine. Le code suivant

```
s = "BCPST"
for x in s:
    print(x)
```

produit le résultat

```
B
C
P
S
T
```

autrement dit la variable `x` prend successivement pour valeurs les caractères de la chaine `s`. On parle d'**itération sur une chaine de caractères**.

Exercice 12
Écrire une fonction `type_lettres(s)` qui prend en argument une chaine de caractères `s`, qu'on suppose formée uniquement de lettres minuscules (un mot), et qui affiche pour chacune des lettres le mot `"voyelle"` ou bien `"consonne"`.