TP 2
Conditions et boucles

Aujourd'hui nous écrivons des programmes avant tout dans le mode script (sur plusieurs lignes) ; le mode interactif sert à faire des tests courts ou à inspecter le contenu ou le type des variables. Séparer les programmes et les exercices par des **cellules** délimitées par une ligne commençant par la suite de trois caractères **#%%** permet de tout garder sur une même page (donc d'enregistrer dans un même fichier `.py`), mais de n'exécuter qu'un morceau à la fois et pas tout depuis le début. La présentation du fichier doit donc ressembler à ceci, comme dans les corrections :

```
#%% exercice 1
...

#%% exercice 2
...
```

On utilise alors l'option « exécuter la cellule » (*execute cell*), aussi sur le raccourci clavier Ctrl + Entrée.

I Conditions simples

Le mot-clé `if` suivi d'une condition introduit un morceau de programme qui va être exécuté **si** la condition est vérifiée. Éventuellement, le mot-clé `else` (**sinon**) introduit un morceau de programme qui va être exécuté dans le cas contraire.

Voici un exemple de bout de programme, qu'on peut recopier tel quel et tester :

```
x = int(input("Entrez un nombre : "))
if x >= 0:
    print("positif")
else:
    print("strictement négatif")
print("FIN")
```

Cela est bien entendu un concept fondamental de la programmation, qui permet de rendre un programme interactif et dont le résultat va dépendre des entrées.

La syntaxe générale en Python est la suivante :

```
if condition:
    instructions
else:
    instructions sinon
```

où :

- `condition` désigne n'importe quelle expression booléenne que le programme va tester. Elle est formée notamment, on le rappelle, avec
  - L'égalité : `==`
  - La différence : `!=`
  - Les comparaisons strictes : `<`, `>`
  - Les comparaisons larges : `<=`, `>=`
  - Les mots-clés `and`, `or`, `not`
- `instructions` est du code Python, qui peut être sur plusieurs lignes, qui va être exécuté seulement si la condition a été évaluée à `True`. La partie du programme qui va être exécutée est tout ensemble décalée vers la droite (on dit **indentée**). En général le décalage est de 4 espaces, ou un seul caractère tabulation ; le logiciel sert notamment à bien aligner les lignes. Elle forme un **bloc d'instructions**.
- `instructions sinon` est un **autre bloc d'instructions** qui va être exécuté dans le cas contraire.

- Ensuite, le code qui n'est plus décalé vers la droite ne fait plus partie des blocs d'instructions ; il est donc exécuté dans tous les cas. Le `else` n'est d'ailleurs pas du tout obligatoire : si la condition est fausse alors le premier bloc d'instruction n'est pas exécuté et le programme passe directement à la suite.

On n'oubliera pas non plus le symbole double points, qui fait partie de la syntaxe, termine la ligne et introduit le bloc d'instructions. Il permet aussi au logiciel de proposer directement d'indenter, le saut de ligne après le double point décale automatiquement à droite et on n'a rarement besoin d'écrire à la main les espaces ou tabulations.

Exercice 1
Écrire un programme qui demande à l'utilisateur son âge, et affiche s'il est majeur ou mineur.

Exercice 2
Écrire un programme qui demande à l'utilisateur un nombre, et affiche sa valeur absolue.

Exercice 3
Choisissez un mot de passe secret ; écrire un programme qui demande à l'utilisateur d'entrer un mot, et qui lui dit si c'est le bon mot de passe ou non.

Le bloc d'instructions peut lui-même contenir d'autres conditions emboitées. L'indentation des blocs est alors cruciale.

Exercice 4
Améliorer programme d'exemple ci-dessus pour demander à l'utilisateur un nombre et afficher s'il est positif, négatif ou nul. Sans regarder la suite du TP.

II Conditions en cascade

Il arrive que l'on veuille tester une condition plus complexe qui ne se traduit pas aussi simplement que « si… alors ». Le mot-clé `elif` est la contraction de « else, if » (**sinon, si**) et introduit une condition qui va être testée si la précédente était fausse, ainsi qu'un bloc d'instruction correspondant à exécuter. Par exemple le programme du dernier exercice peut se ré-écrire ainsi :

```
x = int(input("Entrez un nombre : "))
if x > 0:
    print("positif")
elif x < 0:
    print("négatif")
else:
    print("nul")
```

Il est tout à fait possible d'enchainer plusieurs `elif` ; les conditions sont simplement testées les unes à la suite des autres. Le `else` final capture le cas où **aucune** des conditions n'a été vérifiée ; il n'est toujours pas obligatoire, si aucune condition n'est vérifiée le programme passe simplement à la suite. La syntaxe générale ressemble donc à :

```
if condition1:
    instructions1
elif condition2:
    instructions2
elif ... :
    ...
else:
    instructions sinon
```

Ainsi lors de l'exécution :
- Le programme teste la `condition1`. Si elle est vraie alors il exécute `instructions1`.
- Sinon, il vérifie la `condition2`. Si elle est vraie alors il exécute `instructions2`.
- Et ainsi de suite.
- À la fin, si aucune condition n'a été vérifiée, le programme exécute le bloc d'instructions du `else`.

Exercice 5
Écrire un programme qui demande son âge à l'utilisateur, et affiche s'il est majeur, mineur, ou senior (plus de 65 ans).

Exercice 6
Écrire un programme qui demande à l'utilisateur sa note au bac (attention, ce n'est pas forcément un nombre entier) et affiche à quelle mention cela correspond. Vous pouvez l'assortir librement d'un commentaire sur la note…

À retenir

Les conditions en cascade sont testées **successivement** quand la précédente a échoué. À cause de cela, il y a un ordre logique et naturel dans lequel on doit écrire ses conditions. De plus le mot `else` n'est pas suivi par une condition (c'est un « sinon » tout court) et il s'exécute quand aucune des conditions précédentes n'a été vérifiée.

III Boucles

Une **boucle** permet de répéter des instructions automatiquement. C'est à partir de maintenant que les programmes deviennent *vraiment* intéressants : ils automatisent des tâches qui seraient bien pénibles pour un humain.

Dans ce TP on se concentre sur la boucle `while`. Le bloc d'instructions (**corps** de la boucle) est répété **tant que** (traduction de *while*) une condition est vérifiée. La syntaxe est tout simplement la suivante :

```
while condition:
    instructions
```

alors lors de l'exécution :
- Le programme teste si la condition est vraie,
- Si oui, il exécute le bloc d'instructions. Une fois fini, il recommence à évaluer la condition et ainsi de suite.
- Sinon, il sort simplement de la boucle et passe à la suite.

Pour que cela soit intéressant, il faut que la condition puisse varier à chaque passage dans la boucle ! Sinon, si elle est toujours vraie, rien ne l'arrête et on obtient une **boucle infinie**… La méthode de base pour répéter une instruction un certain nombre n de fois est de déclarer une variable appelée **compteur**, que l'on va augmenter de 1 (on dit **incrémenter**) à chaque passage dans la boucle, et de tester la condition `i < n` :

```
i = 0
while i < 3:
    print("i =", i)
    i = i + 1
print("fin avec i =", i)
```

produit le résultat suivant :

```
i = 0
i = 1
i = 2
fin avec i = 3
```

Avertissement

Dans chaque boucle, on porte une attention particulière à :
- La valeur à laquelle le compteur est initialisé (essayez avec `i = 1`)
- L'utilisation de la comparaison stricte `<` ou large `<=` (essayez de remplacer par `<=`)
- L'incrémentation au début ou à la fin de la boucle (essayez d'échanger les deux lignes dans le bloc d'instructions),
- Ne pas oublier l'incrémentation (essayez de l'enlever !)

L'exemple ci-dessus est le plus standard et est écrit de telle façon à ce qu'il répète exactement `n` fois.

Exercice 7
Écrire un programme qui affiche les nombres x^2 pour $1 \leqslant x \leqslant 10$.

Exercice 8
Écrire un programme qui demande à l'utilisateur un nombre `n` et compte à rebours : affiche n puis $n-1$ puis… jusqu'à 0. Il y a deux façons possibles, tester les deux :

1. Une boucle croissante comme ci-dessus, où on incrémente `i`, mais en affichant la quantité `n - i`.
2. Une boucle décroissante, où on décrémente `i` (c'est-à-dire le faire diminuer de 1 avec l'opération `i = i - 1`), mais alors il faut changer la condition dans le `while` ainsi que la valeur initiale de `i`.

IV Application aux suites

Pour calculer les termes successifs d'une suite, on se sert en plus d'une variable `u` qui à chaque passage dans la boucle va devenir le terme suivant. L'exemple de base pour les puissances de 2 est :

```
u = 1
i = 0
while i < 5:
    print(u)
    u = 2 * u
    i = i + 1
```

qui produit l'affichage

```
1
2
4
8
16
```

autrement dit les 5 premières puissances, soit de 2^0 à 2^4.

Exercice 9
Écrire un programme qui demande à l'utilisateur un nombre `n` et calcule la somme $1 + 2 + \cdots + n$.

L'intérêt de la boucle `while`, c'est aussi de pouvoir s'arrêter quand une certaine condition sur la suite devient vérifiée — c'est à dire la continuer **tant que** elle n'est **pas** vérifiée.

Exercice 10
Au début de l'année 2026, la population mondiale est estimée à environ 8,30 milliard d'habitants. Elle augmente d'environ 0,84 % chaque année. Écrire un programme qui affiche la population mondiale estimée sur les années futures (afficher à la fois l'année et la population, par exemple `Année 2026 : 8300000000`) si le taux de croissance reste le même, et s'arrête quand elle dépasse 10 milliards.

Exercice 11 (*)
La **suite de Syracuse** est la suite entière $(S_n)_{n\in\mathbb{N}}$ définie par : le nombre $S_0 \geqslant 1$ est à déterminer par l'utilisateur, et ensuite

$$S_{n+1} = \begin{cases} S_n/2 & \text{si } S_n \text{ est pair} \\ 3 \times S_n + 1 & \text{si } S_n \text{ est impair} \end{cases}$$

1. Que se passe-t-il si $S_0 = 1$?
2. Écrire un programme qui demande à l'utilisateur le nombre S_0 puis affiche tous les termes de la suite jusqu'à ce qu'un terme soit égal à 1.
3. La célèbre **conjecture de Syracuse** est l'énoncé selon lequel quelque soit le nombre S_0, la suite finit par retomber sur 1. Il s'agit d'un problème ouvert célèbre…

   Écrire un programme qui nous aide à vérifier cette conjecture, en demandant à l'utilisateur un entier N puis en testant la conjecture pour tout $1 \leqslant S_0 \leqslant N$, affichant pour chaque valeur testée si la conjecture est bien vraie.
4. En plus, pouvez-vous compter, pour chaque S_0, en combien d'étapes la suite revient à 1 ?

On peut vérifier la conjecture pour des valeurs extrêmement grandes, mais ceci ne constitue malheureusement toujours pas une démonstration mathématique…