TP 4
Divers compléments (`for`, `tuple`, `assert`)

I Rappel des épisodes précédents

Les types vus jusque là :
- `int` : nombres entiers
- `float` : nombres à virgule flottante (virgule `.` et notation scientifique avec `e`)
- `str` : chaines de caractères, entre guillemets doubles `"`
- `bool` : booléens, avec `True` et `False`
- Connaitre le type : fonction `type()` (mode interactif)
- Conversions de type : fonctions `int()`, `float()`, `str()`, `bool()`
- Remarque : les fonctions sont aussi des objets du type... `function`

Les opérations :
- Les 4 opérations `+`, `-`, `*`, `/`
- Le quotient dans la division euclidienne (en nombres entiers) `//`, le reste `%`
- Les comparaisons `==`, `!=`, `<`, `<=`, `>`, `>=`
- Les opérations sur les booléens `and`, `or`, `not`
- Les opérations sur les chaines de caractères :
  - concaténation `+`
  - multiplication par un nombre entier `*`
  - accès au `i`-ème caractère de `s` (démarrant à `0`) par `s[i]`
  - longueur de la chaine `len(s)`

Les fonctions :
- Afficher : `print()`
- Demander à l'utilisateur : `input()`, mais **nous ne l'utilisons plus** car nous écrivons des fonctions avec des paramètres.
- Aide sur une fonction ou un module : `help()` (mode interactif)

Les conditions (les `elif` et le `else` ne sont pas du tout obligatoires) :

```
if condition1:
    instructions1
elif condition2:
    instructions2
elif condition3:
    instructions3
...
else:
    instructions
```

Les boucles :

```
while condition:
    bloc d'instructions
```

Les déclarations de fonctions :

```
def nom(argument1, argument2, ...):
    """ documentation """
    bloc d'instructions
```

avec l'instruction `return` qui arrête l'exécution et éventuellement renvoie une valeur.

*On rappelle qu'à partir de maintenant on écrit essentiellement des **fonctions** avec des paramètres et une valeur de retour. Un programme est donc présenté comme une suite de fonctions, séparées par des cellules `##` ; exécuter le programme va enregistrer les fonctions mais ne va rien afficher. Les tests, eux, se font dans le mode interactif (sauf ceux nécessitant plusieurs lignes, par exemple avec une boucle et des `print` dedans).*

II La boucle `for`

Dans sa syntaxe de base, la boucle `for` permet de répéter un bloc d'instructions où une variable augmente de un en un. Ainsi la boucle

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

est **exactement équivalente** à

```
i = a
while i < b:
    print(i)
    i = i + 1
```

Le fait qu'on utilise un signe d'inégalité stricte `<` signifie que **la boucle s'arrête avant la deuxième borne du `range`**. Par exemple `range(0, 10)`, qui se note aussi note simplement `range(10)`, répète exactement 10 fois, et il y a bien 10 nombres dans $0, 1, \dots, 9$.
On l'utilise quand on sait à l'avance combien de fois on va répéter la boucle, mais qu'on ne veut pas l'arrêter en cours de route comme `while`.
Cependant le mécanisme de la boucle `for` en Python est nettement plus complexe. Par exemple pour parcourir une chaine de caractères et afficher tous les caractères uns par uns, on sait qu'on peut écrire

```
s = "Bonjour"
longueur = len(s)
for i in range(longueur):
    print(s[i])
```

(Remarquez que la chaine ici est de longueur 7, donc son premier caractère est `s[0]` et son dernier est `s[6]` : cela est cohérent avec le fait que `range(7)` fait varier `i` entre `0` et `6` inclus. Souvent on écrit directement `for i in range(len(s))`)
Mais ici on peut aussi écrire :

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

de telle façon que `x` prend successivement pour valeur chacun des caractères de `s`. On dit qu'on **itère sur les éléments** au lieu d'**itérer sur les indices**. Cela est pratique à la condition… que le corps de la boucle utilise seulement la valeur de `x` mais n'a pas besoin de connaitre son indice.
D'ailleurs, qu'est-ce que `range` ? **Tester** (mode interactif)

```
type(range(10))
3 in range(10)
12 in range(10)
```

Cela est un *type intervalle* et la syntaxe de la boucle `for` ressemble beaucoup à un $\forall i \in [a, b[\dots$
Si on n'utilise même pas du tout l'indice dans la boucle, on peut aussi ne pas du tout lui donner de nom avec `_` : **tester**

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

Exercice 1. Écrire une fonction `compte_voyelles(s)` qui prend en argument une chaine de caractères `s` et compte le nombre `n` de voyelles dans `s`, en itérant sur les éléments et pas sur les indices.

Exercice 2. Écrire une fonction `table_multiplication()` qui ne prend pas d'arguments et qui affiche tous les produits $n \times m$ pour $1 \leqslant n \leqslant 10$ et $1 \leqslant m \leqslant 10$, en utilisant une boucle `for` imbriquée dans une autre boucle `for`.
Bonus : rendre cela un peu plus joli, par exemple
— Afficher `n` et `m` aussi, par exemple affiche sous la forme `3 * 7 = 21`…

— Afficher la table sous forme d'un carré. Cela nécessite, à certains endroits de la boucle, de demander à `print` de *ne pas* revenir à la ligne après l'affichage, ce qu'on peut faire avec la syntaxe `print(... , end="")` (« lors de l'affichage, remplacer le caractère final (par défaut, un retour à la ligne) par la chaine vide `""` »).
— Aligner : `print("... {0:4d} ...".format(x))` permet d'insérer, dans le texte, le nombre entier `x` en prenant la place de 4 caractères et en l'alignant vers la droite. Cela s'appelle le *formatage* des chaines de caractères et il existe des dizaines de codes de format à combiner, lire tout seul la documentation...

III Le type `tuple`

Cela correspond directement à la notion de produit cartésien en mathématiques. Par exemple la variable

```
t = (4, 12)
```

contient deux nombres entiers et son type est... `tuple`. On peut récupérer les deux valeurs en même temps par

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

qui va affecter deux variables `x`, `y`. On peut aussi avoir plus de deux éléments, séparés par des virgules, et les éléments ne sont pas nécessairement des entiers mais peuvent être de tous les types vus avant.
Une application est d'écrire des fonctions qui retournent plusieurs variables en même temps : on retourne en fait un `tuple` !

Exercice 3. Écrire une fonction `minmax(a, b)` qui prend en argument deux nombres `a`, `b` et retourne le `tuple` formé du minimum et du maximum.

Exercice 4.
1. On représente les coordonnées des vecteurs du plan par un `tuple` de deux nombres. Écrire une fonction `somme(u, v)` qui prend en argument deux `tuple` représentant des vecteurs $\vec{u}$, $\vec{v}$ et retourne un `tuple` correspondant au vecteur $\vec{u}+\vec{v}$.
2. Écrire de même la fonction `produit(a, u)` qui prend en argument un `tuple` correspondant à un vecteur $\vec{u}$ et un nombre `a` correspondant à un réel a et qui retourne un `tuple` représentant le vecteur $a\vec{u}$.
3. En utilisant et en combinant simplement les deux fonctions précédents, écrire une fonction `combinaison_lineaire(a, u, b, v)` (où `u`, `v` correspondent à des vecteurs et `a`, `b` à des nombres) et qui retourne un `tuple` représentant le vecteur $a\vec{u}+b\vec{v}$.

Remarquer que la syntaxe

```
(x, y) = (3, 6)
```

déclare à la fois une variable `x` valant `3` et `y` valant `6`, et la syntaxe

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

va simplement échanger les valeurs de `x` et `y`.

Exercice 5. Rappelons que dans l'exercice 12 du TD 2 nous avons appris à programmer la *suite de Fibonacci* : c'est la suite $(F_i)_{i\in\mathbb{N}}$ avec $F_0=0$, $F_1=1$, et $\forall i\in\mathbb{N}$, $F_{i+2}=F_{i+1}+F_i$. La correction proposée était la suivante :

```
n = int(input("n ? "))
i = 0
a = 0 # représente u_(i)
b = 1 # représente u_(i+1)
while i < n:
    print(a)
    # a devient u_(i+1)
    # b devient u_(i+2) donc c'est u_(i) + u_(i+1) ...
    # ... mais cela nécessitait de sauvegarder u_(i) avant dans c
    c = a
    a = b
    b = c + b
    i = i + 1
```

Le but est de ré-écrire cela de la façon la plus simple et élégante possible :
— C'est une fonction `fibonacci(n)` et elle affiche avec `print` les `n` premiers termes de la suite,
— L'itération se fait avec une boucle `for` et un `range`,
— Une variable `(a, b)` de type `tuple` représente, à chaque passage dans la boucle, le couple (F_i, F_{i+1}).

De plus, il y a une fonction de conversion de type associée `tuple()`. **Tester** par exemple :

```
tuple(range(1, 10))
tuple("Bonjour")
```

Avec la boucle `for`, il est possible d'itérer sur les éléments d'un `tuple` :

```
for x in (1, 3, 5):
    print(x)
```

Bien sûr, un `tuple` peut contenir n'importe quel type de donnée. **Tester** :

```
for x in ("mathématiques", "physique-chimie", "SVT"):
    print("J'aime le cours de", x, "car je suis en BCPST.")
```

IV Débugger et sécuriser un programme

Quand un programme Python est valide pour s'exécuter, mais ne fait pas ce qu'on voudrait, il faut débugger le programme. Une méthode courante est de rajouter beaucoup de `print` intermédiaires pour vérifier le contenu des variables au fur et à mesure. En Python on peut utiliser `print` sur tous les types qu'on connait, notamment sur les `tuple` directement...
Dans le TP 2 nous avons appris à compter à l'envers avec *par exemple* le programme suivant :

```
def compte_rebours(n):
    i = n
    while i != 0:
        print(i)
        i = i - 1
```

Exercice 6. Que se passe-t-il si l'utilisateur rentre `compte_rebours(-1)` ?

Grâce au `print` on détecte rapidement le problème, alors que s'il n'y en avait pas on verrait seulement le programme tourner sans s'arrêter...
La fonction `assert()` prend en argument une condition (expression booléenne) et, **si elle n'est pas satisfaite** alors l'exécution du programme s'arrête et un message d'erreur s'affiche. Ici le mot anglais *assert* a le sens de « affirmer que » : on affirme que la condition doit être vraie pour pouvoir continuer l'exécution du programme. **Tester** le programme :

```
def compte_rebours(n):
    assert(n > 0)
    i = n
    while i != 0:
        print(i)
        i = i - 1
```

et ré-essayer à nouveau !
En fait, on peut aussi renvoyer un message d'erreur avec `assert` (la place de la virgule n'est pas une erreur) : remplacer la ligne et retester

```
assert(n > 0), "n doit être positif !!!"
```

Dans les exercices suivants, il y a des assertions (pas toujours écrites explicitement) à identifier et à faire vérifier par le programme. La vérification doit se faire dès les premières lignes et avant *tout affichage. Sinon, l'examinateur se fera un plaisir de faire échouer le programme.*

Exercice 7. Écrire la fonction `moyenne_harmonique(x, y)` qui prend deux nombres `x`, `y` et calcule leur *moyenne harmonique* : l'unique nombre m tel que

$$\frac{1}{m} = \frac{1}{x} + \frac{1}{y} \tag{1}$$

Généraliser : la fonction prend cette fois comme argument un `tuple` de longueur n quelconque $(x_0, \dots, x_{n-1})$ et itère sur le `tuple` pour calculer le nombre m tel que

$$\frac{1}{m} = \frac{1}{x_0} + \cdots + \frac{1}{x_{n-1}} \tag{2}$$

Exercice 8. Écrire une fonction `ADN(s)` qui prend en argument une chaine de caractères `s` qui doit être composée uniquement des lettres A, T, G, C, parcourt la chaine et affiche pour chaque caractère rencontré le nom en français du nucléotide correspondant.

Remarque 1. Pour une chaine de caractères (ou même pour un seul caractère) `s` il existe diverses fonctions telles que `s.isalpha()` (teste si la chaine est constituée uniquement de *lettres*), `s.isdecimal()` (uniquement les chiffres de 0 à 10), `s.islower()` (tout est en minuscule) et `s.isupper()` (majuscule)… elles renvoient une valeur booléenne et donc peuvent très bien se combiner avec `assert()`.

Exercice 9. Écrire la fonction `bonjour(nom)` qui prend un argument une chaine de caractères représentant un prénom et affiche `"Bonjour"` suivi du prénom… qui doit commencer par une majuscule et ne contenir que des lettres…

V D'autres exercices

Exercice 10. Écrire une fonction `pythagore(N)` qui cherche toutes les solutions (u, v, w) à l'équation

$$u^2 + v^2 = w^2, \quad (u, v, w) \in \mathbb{N}^3 \tag{3}$$

et les affiche. Comme il peut y avoir une infinité de solutions, avec des nombres de plus en plus grand, l'argument `N` sert à limiter la recherche à $u, v, w \leqslant N$.

Aviez-vous traité tous les exercices du TP précédent ?

Exercice 11. Écrire une fonction `compte_points(r)` qui prend en argument un nombre réel r supposé positif et qui compte le nombre de solutions $(n, m) \in \mathbb{Z}^2$ vérifiant $n^2 + m^2 \leqslant r^2$.
Bonus : retourner non pas le nombre N de solutions, mais le quotient $\frac{N}{r^2}$. Qu'en pensez-vous ?