TP 3
Fonctions

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()`

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 d'une chaine avec la fonction `len()`

Les fonctions :
- Afficher : `print()`, peut prendre une liste d'expressions ou de variables séparées par des virgules. Très utile pour afficher les étapes intermédiaires d'un programme.
- Demander à l'utilisateur : `input()`, donne toujours un résultat de type `str`. À partir de maintenant **on ne l'utilise plus : c'est de la décoration**.

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:
    instructions
```

Aujourd'hui et à partir de maintenant nous écrivons des **fonctions** dans le mode script et nous les testons dans le mode interactif ; ainsi exécuter le programme enregistrera les fonctions mais n'affichera jamais rien. Autant que possible les fonctions ne font pas de `print()` ni de `input()` : elles ont des paramètres et des valeurs de retour et c'est l'utilisateur qui décide ce qu'il en fait et comment il voudra les afficher. À nouveau une bonne pratique est de les séparer par des cellules (ligne commençant par `##`).

II Cours

II.1 Déclaration de fonction

La syntaxe générale de la déclaration d'une fonction est la suivante :

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

Où :

— *nom* est le nom de la fonction, de même que pour les noms de variables.
— *arguments1*, *arguments2*, sont des noms de variables, que la fonction pourra utiliser. Il peut y avoir plusieurs arguments séparés par des virgules, ou un seul, ou même aucun (dans ce cas la fonction n'a pas de paramètres et donc fait toujours la même chose ; ce n'est pas inutile pour autant). Ils peuvent contenir n'importe quel type vu jusque là.
— *bloc d'instructions* est, comme dans les conditions et les boucles, tout un bloc d'instructions (indenté à droite) qui constitue le **corps** de la fonction.

Dans la suite du programme, on peut **appeler** la fonction en lui **passant des arguments** avec la syntaxe que nous avons déjà vu pour les fonctions telles que `print()` : par exemple si on a

```
def joliesomme(x, y):
    print(x, "+", y, "=", x+y)
```

alors `joliesomme(3, 4)` va exécuter la fonction nommée `joliesomme` et son premier argument `x` va prendre la valeur `3`, son second `y` la valeur `4`.

II.2 L'instruction `return`

Dans le bloc d'instructions, lorsque le programme tombe sur une ligne commençant par `return`, l'exécution de la fonction **s'arrête**. La valeur qui suit est dite **renvoyée par la fonction**. Tester

```
def somme(x, y):
    return x + y
```

alors l'appel

```
s = somme(3 ,4)
```

va calculer `3 + 4` et le renvoyer pour le mettre dans la variable `s`. D'ailleurs, tester

```
type(somme(3, 4)) # c'est bien le nombre 7
type(somme) # alors ?
```

L'instruction `return` n'est pas obligatoire : si l'exécution de la fonction arrive au bout sans avoir rencontré de `return`, elle s'arrête sans renvoyer le valeur.

```
def f():
    print("Cette fonction n'a pas de paramètre ni de valeur de retour ; est-elle utile à la société ?")
```

et tester

```
f()
y = f()
print(y)
type(y)
```

L'instruction `return` permet aussi de simplement terminer une fonction :

```
def divise_proprement(x, y):
    if y == 0:
        print("Il ne faut JAMAIS diviser par 0 !!!")
        return
    return x / y
```

Tester, avec ou sans la ligne du `return` tout seul ! Remarquer que le `else` n'est pas nécessaire car si la condition dans le `if` est vérifée la fonction va se terminer ; ce qui se passe après le bloc d'instruction du `if` a donc automatiquement une valeur de *sinon*...

II.3 Notion de variable locale

Une fonction a bien entendu le droit d'utiliser des variables dans son corps. Cependant, à moins que ces variables soient renvoyées avec un `return`, elles sont détruites à la fin de l'exécution de la fonction et donc ne sont pas accessibles au reste du programme. On parle de **variables locales**. Essayez par exemple :

```
def compte(n):
    i = 0
    while i < n:
        print(i)
        i = i + 1
```

alors tester le programme depuis le mode interactif :

```
compte(10) # OK
print(i) # ???
```

Cela correspond à la notion mathématique de variable non-libre (on dit aussi *liée* ou *muette*) : du point de vue de l'utilisateur, en dehors de la fonction, `i` n'existe pas et il n'a pas le droit de dire « je veux que `i` soit égal à… »

III À vous de jouer !

On répète une fois de plus que, à moins que l'on demande explicitement que la fonction affiche quelquechose, les fonctions ne font pas de `print()` : elles ont des arguments, font des calculs, utilisent éventuellement des variables locales, et renvoient une valeur de retour avec `return`.

Exercice 1. Écrire la fonction `valeur_absolue(x)` qui retourne la valeur absolue de `x`.

Exercice 2. Écrire la fonction `maximum(a, b)` qui retourne le plus grand des deux nombres `a` et `b`.

Exercice 3. Écrire la fonction `partie_entiere(x)` qui prend en argument un nombre de type `float x` supposé positif, de la façon suivante : une boucle `while` cherche le plus grand entier `n` qui est inférieur à `x`. Peut-on améliorer la fonction pour traiter aussi le cas où `x` est négatif ?

Exercice 4. Écrire la fonction qui calcule la partie entière de la racine carrée sans utiliser la fonction partie entière ni la fonction racine carrée (*Hein ? ? ?*)

IV L'aide

Toutes les fonctions définies en Python possèdent une aide accessible (en mode interactif) avec la commande `help()` :

```
help(abs)
```

Chacun peut créer de la documentation pour sa propre fonction en insérant juste après la déclaration du texte entre trois guillemets doubles successifs `"""`, ce que Python appelle une *docstring* (chaine de documentation), et qui peut tenir sur plusieurs lignes :

```
def mafonction(x):
    """" c'est MA fonction qui calcule le carré de x """
    return x * x
```

Lancer le programme puis essayer en mode interactif :

```
help(mafonction)
```

C'est d'ailleurs une spécificité de Python de pouvoir intégrer l'aide et la documentation d'une fonction directement dans le programme (et pas dans un fichier de documentation à part). C'est une bonne pratique de documenter son propre programme pour que d'autres puissent l'utiliser.

Exercice 5. Remplir soigneusement les documentations des fonctions de la section III.

V Les modules

Un *module* (ou aussi : bibliothèque, librairie) est un ensemble de fonctions. En général, elles sont groupées par thème et permettent de donner des nouvelles possibilités au langage. Les modules Python de base permettent de trouver les fonctions mathématiques, les nombres aléatoires, l'écriture dans des fichiers, mais aussi de connaitre la date, se connecter en réseau, traiter des images… cela contribue au succès de Python d'avoir des milliers de bibliothèques disponibles et d'interagir avec autant de situations.
Le module se charge avec la commande `import`, qui a plusieurs syntaxes. Par exemple, le module `math` comporte beaucoup de fonctions mathématiques comme la fonction exponentielle `exp()`, mais on ne peut pas l'utiliser tout de suite. En général la ligne d'import du module est au début du programme *avant* les cellules des exercices (ou en mode interactif) :

```
# syntaxe 1 : on importe tout
import math
# math.exp(x), math.sin(x), math.pi ...
```

```
# syntaxe 2 : on importe seulement les fonctions nécessaires
from math import exp, sin
# exp(x), sin(x), mais pas pi
```

Il est courant aussi de donner un alias au nom de module quand il est trop long. On peut aussi donner un alias aux fonctions qu'on importe.

```
# très courant... pyplot est un sous-module de matplotlib
import matplotlib.pyplot as plt
# plt.plot(), plt.show() ... au lieu de matplotlib.pyplot.plot(), matplotlib.pyplot.show() ...

# ou encore...
from math import exp as lafonctionexponentielle
# lafonctionexponentielle(x)
```

On trouve aussi la déclaration suivante :

```
from math import *
```

qui importe d'un coup toutes les fonctions mathématiques, sans le préfixe `math`, mais attention... La problématique est toujours la suivante : le créateur du module peut créer des tas de fonctions avec des noms courts, il faut donc qu'elles soient précédées du nom du module pour bien les distinguer des autres fonctions créées par l'utilisateur, mais cela donne des noms longs. Cependant si l'utilisateur avait déjà des fonctions avec le même nom que celles dans le module `math` — et il ne sait pas forcément à l'avance tout ce qu'il y a dedans — alors elles seront remplacées par celles du module.

```
pi = 3 # c'est ma variable pi, pourquoi pas ?
print(pi) # OK
from math import *
print(pi) # aïe
```

Un module possède aussi une documentation, accessible une fois chargé :

```
import math
help(math)
```

Remarque 1.
- — On donne ici diverses syntaxes ; le but est d'être capable de les reconnaitre et les interpréter, pas de tout savoir par cœur. En général, dans un sujet de concours écrit, on demande souvent « écrire une fonction qui... » mais quand il s'agit d'utiliser un module, on donne dès le départ la commande précise à utiliser pour charger le module, et les fonctions qu'on utilisera dans le module. Par exemple on dira « on suppose qu'on a importé le module `math` avec la commande `import math as m` » et donc il faut savoir que les fonctions s'appellent alors `m.exp(x)`, `m.sin(x)`...
- — Aux épreuves orales le candidat peut être devant un ordinateur avec le logiciel Pyzo. Si bien entendu il ne pourra pas accéder librement à tout l'internet, utiliser l'aide intégrée de Python est parfaitement légal (et là encore, il ne s'agit pas de tout savoir par cœur sur le langage Python et toutes ses bibliothèques). Il est donc important de comprendre tôt le mécanisme de fonctionnement de Pyzo, du mode interactif, de l'aide et de la documentation fournis, et savoir retrouver rapidement l'aide sur les commandes que l'on aurait oublié.

Exercice 6. Lire la documentation du module `math`, et éventuellement des fonctions contenues dedans, et trouver :

1. La fonction racine carrée,
2. La fonction partie entière,
3. Les PGCD (plus grand commun diviseur) et PPCM (plus petit commun multiple).

... d'ailleurs, y a-t-il plusieurs parties entières ? Quelle différence avec la conversion de type `int()` ?

VI D'autres exercices

Exercice 7. Écrire la fonction `factoriel(n)` qui calcule le produit $1 \times 2 \times \cdots \times n$.

Exercice 8.
1. Écrire une fonction `sommes_carres_impairs(n)` qui calcule la somme $S_n = 1^2 + 3^2 + \cdots + (2n+1)^2$.
2. Écrire une fonction `conjecture(n)` qui retourne `True` si S_n est bien égal à $P(n) = \frac{(n+1)(2n+1)(2n+3)}{3}$ et `False` sinon. Peut-on ainsi démontrer $\forall n \in \mathbb{N}$, $S_n = P(n)$?

Exercice 9. Écrire une fonction `nombre_voyelles(s)` qui pend en argument une chaine de caractères, supposée ne contenir que des caractères en minuscule, et retourne le nombre de voyelles.

Exercice 10. Écrire une fontion `premier_mot(s)` qui prend en argument une chaine de caractères, supposée contenir des mots séparés par des caractères espace, et qui retourne le premier mot de `s`.
Indication : une variable `m` de type `str` sert d'accumulateur.

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 $n^2 + m^2 \leqslant r^2$ (ce sont les points du plan à coordonées entières contenus dans le disque de rayon r).
Indication : bien borner les variables x, y et utiliser une double boucle.

Exercice 12. Le module `random` contient la fonction `randint(a, b)` qui renvoie un nombre entier aléatoire entre les deux entiers `a` et `b`. Programmer le jeu suivant :

1. Le programme choisit un nombre `x` au hasard et ne le montre pas à l'utilisateur,
2. Il demande à l'utilisateur d'entrer un nombre `y` (ici, on utilise `input()` car on veut vraiment programmer un jeu interactif avec l'utilisateur), et lui affiche si le nombre à deviner est plus petit ou plus grand que `y`,
3. Et il continue, tant que l'utilisateur n'a pas deviné le nombre.

Le programme est contenu dans une fonction qui prend en argument le nombre `n` tel que le nombre à découvrir est entre 0 et n (donc plus n est grand... plus le jeu est difficile) et qui retourne le nombre d'essais qui ont été nécessaires.