TP 3
Fonctions

Aujourd'hui et à partir de maintenant nous écrivons avant tout des **fonctions** dans le mode script. Cela permet de faire varier des paramètres, et de récupérer une valeur calculée.
L'exécution d'un code de déclaration de fonction ne produit aucun affichage, il faut donc tester les fonctions dans le mode interactif ou bien directement à la suite du script. On continue à séparer les exercices par des cellules avec les trois caractères `#%%`.

I Notion de fonction

I.1 Déclaration et appel

Une fonction correspond à un morceau de programme ré-utilisable, dans lequel on peut faire varier des paramètres. Elle se déclare avec le mot-clé `def`. Étudions l'exemple suivant :

```
def somme(x, y):
    print("La somme de", x, "et de", y, "est", x+y)
```

Ce code est une **déclaration** de fonction, son exécution ne produit rien mais enregistre la fonction.
Les tests suivants en mode interactif permettent de faire varier les paramètres. On parle d'**appel** de la fonction.

```
>>> somme(3, 4)
La somme de 3 et de 4 est 7
>>> somme(-6, 2)
La somme de -6 et de 2 est -4
```

Les variables `x` et `y` s'appellent des **arguments** de la fonction. Une fonction `f` peut avoir un ou plusieurs arguments, ou aucun — dans ce dernier cas l'appel s'écrit juste `f()`. Les arguments peuvent être de n'importe lesquels des types manipulés jusque là.
Ce qui suit le mot-clé `def` forme un bloc d'instructions, exactement comme dans les conditions et les boucles. On parle du **corps** de la fonction. Il peut donc contenir lui aussi des variables, des conditions et des boucles.

I.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 ou l'expression qui suit est dite **renvoyée par la fonction** et on peut alors récupérer sa valeur. Prenons l'exemple

```
def moyenne(x, y):
    return (x + y) / 2
```

alors l'appel

```
>>> m = moyenne(12, 14)
```

1. Appelle la fonction, en donnant les valeurs `x` ← `12` et `x` ← `14`
2. Calcule le résultat de `(x + y) / 2` qui donne le nombre `13.0`,
3. **renvoie** ce résultat, et le met ici dans la variable `m` : `m` ← `13.0`

On peut le vérifier :

```
>>> m
13.0
```

Ce comportement ne serait pas possible si on écrivait `print((x + y) / 2)` au lieu de `return` : la valeur serait bien affichée mais serait ensuite perdue et on ne pourrait pas la récupérer dans une variable. Et il est crucial pour la ré-utilisabilité de la fonction de pouvoir ainsi enregistrer la valeur qu'elle calcule.
L'instruction `return` n'est pas obligatoire. Si l'exécution de la fonction arrive au bout sans avoir rencontré de `return`, elle ne renvoie pas de valeur. Si `return` est présent sans valeur de retour, l'exécution de la fonction s'arrête mais ne renvoie pas de valeur.
L'exemple suivant est donc bien valide

```
def f():
    print("Cette fonction n'a pas de paramètres ni de valeur de retour.")
    print("Est-elle pour autant utile ?")
    return
    print("Ceci, par contre, ne s'affichera jamais.")
```

et son appel ne fait qu'afficher toujours le même texte des deux premières lignes.
Un autre exemple instructif est celui-ci, comme son nom l'indique :

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

Alors :

1. Si l'argument `y` est `0`, on entre dans le premier `if`. Le message d'avertissement s'affiche. Puis la fonction se termine, à cause du `return`.
2. Sinon, le bloc d'instruction du `if` n'est pas exécuté. La fonction calcule `x / y` et renvoie cette valeur.
3. Remarquez que du coup le `else` n'est pas nécessaire ici : l'instruction `return x / y` se produit uniquement dans le cas où `y` est non-nul, sinon elle s'est arrêtée avant !

Le test donne :

```
>>> q = divise_proprement(6, 2)
>>> q
3.0
# q contient bien une valeur, le résultat
>>> q = divise_proprement(5, 0)
Il ne faut JAMAIS diviser par 0 !!!
>>> q
# ici rien ne s'affiche ! q existe mais ne contient pas de valeur !
```

I.3 Notion de variable locale

Une fonction a bien le droit d'utiliser des variables dans son corps, et les arguments eux-mêmes sont des variables à part entière. 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**. Reprenons l'exemple

```
def moyenne(x, y):
    m = (x + y) / 2
    return m
```

alors en mode interactif

```
>>> moyenne(12, 14)
13.0
>>> x
NameError: name 'x' is not defined
>>> m
NameError: name 'm' is not defined
```

Ces variables n'existent plus !
Cela correspond à la notion mathématique de variable non-libre, on dit aussi *liée* ou *muette*.
Cela signifie aussi que l'on peut écrire

```
>>> m = 10
>>> moyenne(12, 14)
13.0
>>> m
10
```

car la variable `m` qui est manipulée à l'intérieure de la fonction n'est pas vraiment la même que celle qui pourrait déjà exister avant.

II À vous de jouer !

Les fonctions suivantes ne font pas intervenir de connaissances en Python autres que celles vues dans les TP précédents, mais la mise en forme est différente. On répète une fois de plus que, à moins d'une demande explicite, les fonctions ne font pas de `print()` ni de `input()` : elles ont des arguments, font des calculs, utilisent éventuellement des variables locales, des conditions et des boucles, et renvoient une valeur avec `return`.

Exercice 1. Écrire la fonction `perimetre_rectangle(a, b)` qui renvoie le périmètre du rectangle de côtés a et b.

Exercice 2. Écrire la fonction `valeur_absolue(x)` qui renvoie la valeur absolue de `x`.

Exercice 3. Écrire la fonction `maximum(a, b, c)` qui renvoie le plus grand des trois nombres entre `a`, `b` et `c`.

Exercice 4.

1. Écrire la fonction `partie_entiere(x)` qui prend en argument un nombre de type `float`, `x`, qui pour l'instant ne marchera que si `x` est positif, et qui calcule sa partie entière de la façon suivante : une boucle `while` cherche le plus grand entier `n` qui est inférieur ou égal à `x`.
2. Bonus : améliorer la fonction pour traiter séparément le cas où `x` est négatif. Attention à bien la tester dans tous les cas, `x` positif ou négatif, entier ou non.

Exercice 5. Écrire la fonction qui calcule la partie entière de la racine carrée sans utiliser la fonction partie entière ni la fonction racine carrée (*Hein ?*)

III L'aide

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

```
>>> 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 `""" documentation """`, ce que Python appelle une **docstring**, contraction de *documentation string* (chaine de documentation), et qui peut même tenir sur plusieurs lignes :

```
def moyenne(x, y):
    """" c'est MA fonction qui calcule la moyenne de x et de y """
    return (x + y) / 2
```

Lancer le programme puis essayer en mode interactif :

```
>>> help(moyenne)
```

La possibilité d'intégrer la documentation dans le corps même de la fonction, de naviguer dans l'aide en mode interactif, et la grande qualité de la documentation Python déjà existante, contribuent pour beaucoup à la diffusion de ce langage et à sa facilité d'apprentissage. C'est une bonne pratique de documenter son programme pour que d'autres puissent l'utiliser.

Exercice 6. Remplir soigneusement les documentations des fonctions de la section II.

IV Les modules

Un *module* (ou aussi : bibliothèque, librairie) est un ensemble de fonctions. 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, communiquer en réseau, traiter des images... Cela contribue au succès de Python d'avoir des milliers de bibliothèques disponibles et d'interagir dans autant de situations.
Un module se charge (une fois pour toute !) avec la commande `import`, écrite en mode interactif ou avant le code qui va l'utiliser, qui a plusieurs syntaxes. Prenons pour exemple le module `math` qui comporte beaucoup de fonctions mathématiques comme la fonction exponentielle `exp()`, la fonction sinus `sin()`, et la constante mathématique `pi`. Il n'est pas chargé par défaut, donc on ne peut pas les utiliser tout de suite. On a les possibilités suivantes, avec des différences dans la manière dont sont ensuite nommées les fonctions :

1. Importer tout, et y accéder avec le préfixe :

```
import math
# math.exp(x), math.sin(x), math.pi, ...
```

2. Importer seulement les fonctions dont on a besoin, et y accéder sans préfixe :

```
from math import exp, sin, pi
# exp(x), sin(x), pi
```

3. Il est courant aussi de donner un **alias** au module, introduit avec le mot-clé `as` :

```
import math as m
# m.exp(x), m.sin(x), m.pi, ...
```

   Autre exemple courant : le sous-module `pyplot` du module `matplotlib` est un peu long à écrire

```
import matplotlib.pyplot as plt
# plt.plot(), plt.show(), ...
```

   Cela marche aussi avec les fonctions et variables elles-mêmes, bien que l'utilité en soit discutable...

```
from math import pi as plus_beau_nombre_de_l_univers
print(plus_beau_nombre_de_l_univers)
# 3.1415926535897793
```

4. On trouve aussi parfois la syntaxe

```
from math import *
```

   qui importe d'un coup **toutes** les fonctions du module, sans le préfixe. **Cette syntaxe est déconseillée** car le module peut contenir beaucoup de fonctions et on ne sait pas forcément à l'avance lesquelles... Si l'utilisateur a déjà une fonction `f` et que le module contient lui-même une fonction aussi nommée `f`, alors celle de l'utilisateur sera « désactivée » après l'import et remplacée par celle du module !

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

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

Exercice 7. Importer le module `math` comme ci-dessus, lire sa documentation, et éventuellement celle 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, il y a plusieurs parties entières ? Quelle différence avec la conversion de type `int()` ? Tester avec plusieurs valeurs, entières ou non, positives ou négatives.

Remarque 1.
1. On a donné ici diverses syntaxes ; le but est d'être capable de les reconnaitre et les interpréter, pas de tout savoir par cœur. En général, un sujet de concours qui utilise un module précise comment il est importé, par exemple « 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)`, etc. Si ce n'est pas précisé, alors cela peut être au candidat de ne pas oublier d'écrire la commande d'import !
2. 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.

Exercice 8. Écrire une fonction Python qui correspond à la fonction mathématique

$$f : x \longmapsto \sin^3(2\pi x)e^{\sqrt{x}} \tag{1}$$

Un autre module d'intérêt est le module `random`, lié à tout ce qui est l'aléatoire, qui contient la fonction `randint` générant un nombre aléatoire entier. Nous l'approfondirons en lien avec les chapitres de probabilités.

Exercice 9. Importer uniquement cette fonction, lire sa documentation, et écrire une fonction `dés(n)` qui affiche `n` fois de suite le résultat d'un lancer de dé cubique aléatoire.

V D'autres exercices

La commande spéciale `assert` suivie d'une condition et éventuellement d'un message d'erreur permet de faire échouer tout le programme, en provoquant une erreur (en rouge) et en affichant le message, si la condition **n'est pas** vérifiée. On écrit par exemple `assert x >= 0, "x doit être positif"`. Le verbe anglais *assert* (comme dans *assertion*) se traduit ici avec le sens plus fort de « affirmer que », autrement dit on affirme que `x` doit être positif pour pouvoir continuer. Cela lui donne un sens proche du langage mathématique « supposons $x \geqslant 0$ » — si ce n'est pas le cas alors la suite n'a pas de sens.
Il est courant qu'une fonction démarre par une ou plusieurs assertions qui servent à vérifier si les arguments donnés sont bien valides et évitent de faire des calculs qui provoqueront plus tard une erreur ou donneront des résultats incohérents. Par exemple la fonction `divise_proprement` de la section I.2 peut s'écrire

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

et l'exécution s'affiche comme un message d'erreur Python

```
>>> divise_proprement(5, 0)
AssertionError: Il ne faut JAMAIS diviser par 0 !!!
```

Dans les exercices suivants, vous pouvez utiliser `assert` pour éviter que les programmes donnent des erreurs, des boucles infinies ou des résultats incohérents.

Exercice 10. La *moyenne harmonique* de deux nombres x, y est l'unique nombre H tel que

$$\frac{1}{H} = \frac{1}{x} + \frac{1}{y}. \tag{2}$$

Écrire une fonction `moyenne_harmonique(x, y)` qui renvoie la moyenne harmonique de `x` et de `y`.
Attention, on veut qu'à aucun moment la fonction ne provoque d'erreur de division par zéro, mais qu'elle avertisse l'utilisateur avec un message d'erreur !

Exercice 11. Écrire une fonction `seuil(t)` qui prend en argument un nombre réel `t` et renvoie le plus petit entier $n > 0$ pour lequel

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n} \geqslant t. \tag{3}$$

Tester des valeurs de `t` de plus en plus grandes entre 3 et 20 (doucement, pas à pas !)
Bonus : dans une deuxième fonction, afficher les valeurs de $t - \ln(\text{seuil}(t))$ pour t de plus en plus grand (par exemple pour tout $5 \leqslant t \leqslant 16$ en sautant de 0,2). Que constate-t-on ?

Remarque 2. Il n'est jamais exigé des étudiants d'utiliser `assert`, et c'est même très déconseillé à l'écrit. En effet en algorithmique on considère que le programme est correct seulement s'il l'est quand on lui donne des bonnes valeurs, et c'est l'utilisateur qui est responsable de donner les bonnes valeurs. Par exemple une phrase telle que « écrire une fonction qui prend en argument un réel x supposé positif… » se comprend comme : la fonction doit donner le bon résultat si $x \geqslant 0$, on ne se préoccupe pas du reste. Ce n'est pas tout à fait la même façon de penser que si on écrivait un programme convivial qui devrait gérer toutes les erreurs possibles en prévenant l'utilisateur que ses valeurs sont incorrectes, en lui demandant de recommencer, etc et cela est en fait un problème assez compliqué à gérer.

On rappelle que la divisibilité se teste à partir de l'opération `%`.

Exercice 12. Écrire une fonction `est_premier(n)` qui prend en argument un entier n en supposant $n \geqslant 2$ et qui renvoie `True` si n est premier et `False` sinon, en tentant de diviser n par tous les entiers inférieurs à n.