Partie 1

Les fonctions en Python

Cours à faire après la partie binaire, représentation des caractères, etc.

1.1 Définitions

1.1.1 Pourquoi et comment on utilise des fonctions

En informatique, une **fonction** est une portion de code représentant un sous-programme, qui effectue une tâche ou un calcul relativement indépendant du reste du programme et qui renvoie une valeur. En Python, la syntaxe est la suivante :

```
def nom_fonction(params):
    instructions
    return résultat
```

Une fonction permet d'**encapsuler** un bloc d'instructions et de lui donner un nom.

Pour définir une fonction, il faut utiliser le mot-clé `def`, puis le nom suivi de parenthèses. Entre les parenthèses, on place des paramètres séparés par une virgule, ou bien rien du tout si la fonction ne demande aucun paramètre.

Exemple de fonction écrite dans l'éditeur :

```
def somme(a, b):
    s = a + b
    return s
```

1.2 Exécuter la fonction

On peut ensuite exécuter ce bloc en utilisant ce nom. On dit qu'on **appelle** cette fonction.

APPELER = ECRIRE LE NOM + () *(sans le def)*

On doit placer autant d'**arguments** qu'il y a de **paramètres** dans la déclaration de la fonction.

Exemple de fonction appelée depuis la console :

```
>>> somme(12, 5)
17
```

1.2.1 En pratique : Où écrit-on le script de la fonction ? À quel endroit de l'IDE appelle-t-on la fonction ?

La fonction peut être définie dans l'éditeur de l'IDE, puis appelée soit : - **Depuis la console/l'interpréteur** de l'IDE (comme Pyzo, Thonny, etc.) - **Depuis le programme lui-même** : dans ce cas, on place les fonctions au début du script, généralement après/avant la déclaration des variables globales. Puis vient le programme principal (le main).

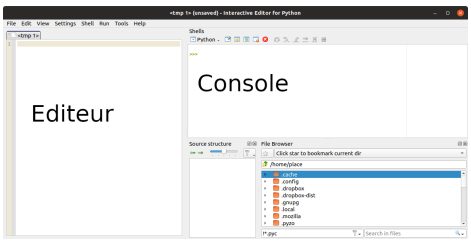

FIGURE 1 – exemple avec Pyzo

1.3 Fonctions natives

Le langage Python contient de nombreuses **fonctions natives**. Par exemple, pour changer le type d'une variable : `int()`, `str()`, `bool()`, `float()`, etc.

Et pour transformer des nombres entre différentes numérations :

```
>>> bin(55)
'0b110111'
>>> int(0b101)
5
>>> int(0x11)
17
>>> hex(55)
'0x37'
>>> hex(0b11111111)
'0xff'
```

1.4 Paramètres et valeur de retour

Une fonction peut avoir un ou plusieurs **paramètres** qui permettent de **transmettre** des valeurs au bloc d'instructions. À l'intérieur du bloc, les paramètres sont traités comme des variables.

Une fonction peut **retourner** une valeur, après l'instruction `return`. Cette valeur retournée peut être un nombre, une chaîne de caractères, un objet, etc.

Récapitulatif :

Étape	Syntaxe	Explication
Création	`def nom(paramètres):` `bloc`	On définit la fonction avec ses paramètres
Appel	`nom(arguments)`	On exécute la fonction en passant des arguments

Lors de l'appel, on dit que l'on **passe un argument**. Par exemple, si on écrit `somme(5, 3)`, on passe les arguments 5 et 3 aux paramètres `a` et `b`.

FIGURE 2 – illustration du passage d'argument lors de l'appel d'une fonction

---

1.5 Fonction mathématique et fonction Python : retourner un nombre

On peut programmer une fonction pour qu'elle retourne la valeur y = f(x).

Exemple : Soit la fonction mathématique f : x ⟶ $3x^2 + 2$

Le script Python correspondant sera :

```
def f(x):
    return 3*x**2 + 2
```

Pour exécuter la fonction avec x = 5, on fait : `f(5)`

Et cela retourne $3 \times 5^2 + 2$, soit 77.

1.6 Retourner une chaîne de caractères

Avec l'exemple vu en TP :

```
def etoiles(nb):
    return '*' + nb * ' ' + '*'
```

On peut appeler cette fonction plusieurs fois :

```
>>> etoiles(1)
'* *'
>>> for i in range(3):
...     print(etoiles(i))
'**'
'* *'
```

```
'*   *'
```

Question : Quelle figure donne le script suivant ?

```
for i in range(4):  # range(4) => i prend les valeurs 0, 1, 2, 3
    print(etoiles(i))
```

1.7 Procédure

Une fonction **sans valeur de retour** est une **procédure**. Elle effectue une action mais ne renvoie rien avec `return`.

Exemple avec le module Turtle (voir TP Turtle).

1.8 Importer des fonctions supplémentaires

En général, cela se fait en important tout ou une partie d'un **module**. On importe alors des fonctions, des objets ou des classes. D'où la notation pointée.

```
from random import randint
for i in range(10):
    print(randint(0, 2))
```

On peut aussi écrire :

```
import random
for i in range(10):
    print(random.randint(0, 2))
```

1.9 Portée des variables

Lorsqu'une fonction utilise des variables, celles-ci sont normalement **propres à la fonction** et ne sont pas accessibles à l'extérieur de celle-ci. On dit qu'il s'agit de **variables locales**, par opposition aux **variables globales** du programme principal.

Exemple : Si on définit une variable `a` avant la définition de la fonction, celle-ci sera différente du `a` de la fonction `somme` :

```
a = 10
def somme(a, b):
    s = a + b
    return s
```

```
>>> somme(1, 0)
1
>>> a
10
>>> b
NameError: name 'b' is not defined
```

Ici, le a global vaut toujours 10, et b n'existe pas en dehors de la fonction.

1.9.1 Modifier une variable globale (à éviter)

Python permet d'affecter à l'intérieur d'une fonction une variable globale. Pour cela, il faut déclarer la variable comme étant `global` au début de la fonction :

```
a = 10
def somme(b):
    global a
    a = a + b
    return a

>>> somme(1)
11
>>> a
11
```

Attention : Modifier une variable globale depuis une fonction s'appelle un **effet de bord**. C'est considéré comme une **mauvaise pratique**, mais parfois indispensable.

1.10 Portée de la fonction, modules

1.10.1 Portée

Une fonction peut être placée dans un autre fichier. On a alors : - Un **fichier principal** (le programme main) - Un ou plusieurs **fichiers annexes** (modules)

Ces fonctions ne deviennent accessibles que si on les **importe**.

Le programme principal doit alors faire référence aux modules avec l'une des manières suivantes :

```
import module
from module import *
import module as alias
```

Il est recommandé toutefois de ne charger que les fonctions utiles du module :

```
from module import fonction
```

Exemple : Supposons que le programme suivant soit dans le fichier `test.py` :

```
# contenu du fichier test.py
a = 10
def somme(b):
    s = a + b
    return s
```

On exécute le programme dans le shell :

```
>>> from test import *
>>> a
10
>>> somme(0)
10
```

Note : Il existe d'autres moyens d'importer une fonction et de l'utiliser en y faisant référence avec une notation pointée.

1.10.2 Complément sur les modules

Def :Un **module** apporte des fonctions et des variables et permet une extension du langage. Il existe différents moyens d'appeler le module et ses fonctions :

import du module	appel d'une fonction du module
import math	math.sin(x)
from math import sin	sin(x)
from math import *	sin(x)
import numpy as np	np.arrange(3, 15, 2)

1.11 Conclusion

Une **fonction** est un morceau de code qui porte un nom et qui s'exécute lorsqu'on l'appelle. L'avantage est d'avoir un code plus court, lisible, mais aussi d'encapsuler les variables et de limiter leur portée à celle de la fonction.

Dans un projet plus grand, les fonctions peuvent être mises dans des **modules** (des fichiers séparés). On doit alors les importer pour avoir une extension du langage. Certains modules très utiles : `math`, `turtle`, `random`, etc.

1.12 Suite de la séance

⇒ TD sur les fonctions à l'écrit (20 min)

⇒ TP Pixel Art (60 min)