INF135 - Fonctions et tests

Hugo Leblanc

Semaine 3

Objectifs de la semaine

- Fonctions
  - Fonction minimale
  - Paramètres d'entrées
  - Retour
- Fonctions vs scripts
  - Portées des fonctions
  - Durées de vies des variables
  - Passage par valeurs
  - Variables globales
- Modules
- Tests

Fonctions

- Les fonctions sont des séries d'instructions contenues dans un ensemble pouvant être facilement re-utilisé. Par convention, la fonction doit faire une tâche simple et précise;
- Contrairement aux scripts, les fonctions n'auront pas de saisit avec **`input`** et pas d'affichage avec **`print`**. Cela à moins d'être expressément demandé.
- Les fonctions peuvent demander des paramètres (ou arguments) d'entrées. Ce sont des informations préalables aux calculs de la fonction;
- Les fonctions peuvent émettre des réponses que l'on appelle retour.

Présentation d'une fonction minimale

```
def nomFonction():
    instructions
```

- Une fonction est au début du fichier;
- La fonction peut ensuite être appelée par son nom d'autres programmes.

Paramètres d'entrées

- Une fonction peut demander des paramètres d'entrées;
- Ceux-ci seront dans une liste après le nom de la fonction entre parenthèses délimitée par une virgule;
- Les paramètres d'entrées seront fournis par l'appelant de la fonction et peuvent donc être utilisés à l'intérieur des instructions de la fonction sans être assignés préalablement.

```
def fcnAvecIn(var1, var2):
    print(var1 * var2)
```

Retours

- La fonction peut retourner un résultat qui sera utilisable par l'appelant de la fonction.
- Le mot-clé **`return`** est utilisé quand l'expression de retour est prête à être renvoyée.
- Une fonction qui ne contient pas de retour est aussi nommée une procédure.

```
def fcnAvecRetour():
    instructions
    return expression
```

Exercice 1

- Écrivez une fonction qui trouve l'aire d'un triangle à partir de sa base et sa hauteur.
- Écrivez une fonction qui détermine si un nombre est impair.

Présentation de l'en-tête d'une fonction

```
def nomFonction(arg1, arg2):
"""
Description générale de la fonction
Args:
  arg1 (float) - Description de l'arg1
  arg2 (bool) - Description de l'arg2
Returns:
  float: Description de retour1
Example:
  >> nomFonction(4,6)
      34
"""
    instructions
```

Contexte et durée de vies des variables

- Tout ce qui se passe à l'intérieur des fonctions est détruit après l'appel de la fonction;
- Toute déclaration de variables à l'intérieur d'une fonction est détruite après l'appel de la fonction;
- Seule la valeur de retour est renvoyée.

Passage par valeurs

- Les paramètres et les retours sont renommés pour la durée de la fonction. Les noms des paramètres de la fonction se nomment paramètres formels;
- Seules leurs valeurs seront transférées entre la fonction et l'appelant. On utilise le terme paramètres effectifs;
- Les noms des paramètres et des retours n'ont aucune incidence;
- L'ordre des paramètres et des retours est ce qui sera considéré.

Exercice 2

- Quel est le résultat de l'affichage du script `passageParValeurTest.py`

```
# passageParValeurTest.py
def  passageParValeur(x, y):
    x = x + 2
    y = y - 2
    z = x - y + 2
    return z

x = 4
z = 8
y = 6
x = passageParValeur(y,x)
print('La valeur de x,y et z sont :', x, y, z)
```

Variables globales

- Une variable globale permet de lier une variable de l'espace de travail normal à l'espace de travail d'une fonction;
- La variable est déclarée comme globale dans l'espace de travail principal du script;
- Sauf avis contraire, l'utilisation de variable globale est **interdite** dans le cours.

Modules

- Plusieurs fichiers de python avec différentes fonctions peuvent interagir entre eux. Chaque fichier créé est considéré comme étant un module qu'on peut importer dans pour nos programmes;
- La commande **`import`** permet d'importer les fonctions d'un autre fichier;
- Le nom du module est le nom du fichier sans son extension `.py`;
- L'utilisation des fonctions du module importer devra être précédée du nom du module et un point.

```
import mon_module # Du fichier mon_module.py

# Appel de la fonction du module importé.
mon_module.ma_fonction(4, 5)
```

Modules fournis

- Plusieurs modules sont fournis et nous verrons qu'il existe un très grand écosystème de module pouvant nous faciliter la création de nos programmes.
- Par exemple, le module `math` nous donne plusieurs fonctions mathématiques communes.
- Pour savoir lesquelles et leurs utilisations, il faut consulter leur documentation;
    - https://docs.python.org/3/library/math.html

```
import math

reponse = math.sqrt(45)
```

Exercice 3

- Écrivez une fonction qui reçoit la base et hauteur et retourne la valeur de l'angle adjacent à la base d'un triangle isocèle.

Tests

- Pour assurer la bonne opération de nos fonctions, des tests seront implémentés;
- Nous utiliserons la libraire `pytest` pour faire nos tests. Il existe d'autres librairies de tests.
- Un test est un script qui, par convention, utilise le nom du programme à tester suivi du mot `test`;
- Le test va convenir des fonctions débutant par le mot `test_` qui seront utilisées par `pytest`;
- La fonction `assert` teste une condition. La fonction ne fait rien si la condition est vraie et génère une erreur si la condition est fausse.
- Le script de tests utilise la fonction `assert` pour faire des appels de la fonction à tester avec des paramètres d'entrées arbitraires;
- Les fonctions de tests n'ont pas de commentaires.

Exemple de tests

```python
# Tests pour des fonctions du module math
import math

def test_factorial():
    assert math.factorial(4) == 24
    assert math.factorial(20) == 2432902008176640000

def test_gcd():
    assert math.gcd(8, 6) == 2
```

Exercice 4

- Écrivez des scripts de tests pour les fonctions de calcul d'aire et de détection de nombre impair.

Tests de valeurs fractionnaires

- Les valeurs fractionnaires sont plus difficiles à tester, car la comparaison de valeurs fractionnaires est assez difficile;
- Pour tester des valeurs fractionnaires, des valeurs différentielles absolues seront utilisées pour faire le test ou le module `pytest` nous fournit la fonction `approx`.

```
import pytest
import math

% Tests de la fonction sin
assert math.fabs(math.sin(1) - 0.8415) < 0.0001
assert math.fabs(math.sin(2) - 0.9093) < 0.0001
assert math.fabs(math.sin(1.5) - 0.9975) < 0.0001

assert math.sin(1) == pytest.approx(0.8414709848)
```