Python – Perfectionnement

Nassib Abdallah

Sommaire :

- **Fonctions.**
- **Fonctions Récursives.**
- **Gestion des exceptions.**
- **Gestion des répertoires et des fichiers en Python.**
- **Fonctions Lambda.**
- **Fonctions Map et Filter.**
- **Listes en compréhension.**
- **Projet**

Fonctions

- **Qu'est-ce qu'une fonction en Python ?**
  - En Python, une fonction est un groupe d'instructions qui accomplit une tâche spécifique.
  - Une fonction est un bloc de code qui ne fonctionne que lorsqu'il est appelé.

- **Pourquoi ?**
  - Les fonctions aident à diviser notre programme en morceaux plus petits. Au fur et à mesure que notre programme s'agrandit, les fonctions le rendent plus organisé et plus facile à gérer.
  - elles évitent les répétitions et rendent le code réutilisable.

Fonctions

- ## Syntaxe de la fonction :

1. Le mot-clé **def** : le début de l'en-tête de la fonction.
2. Un **nom de fonction** pour identifier la fonction de manière unique.
3. Les **paramètres** (arguments) par lesquels nous passons des valeurs à une fonction. Ils sont facultatifs.
4. **Deux points** (:) pour marquer la fin de l'en-tête de la fonction.
5. Une ou plusieurs **instructions** python valides qui composent le corps de la fonction. Les instructions doivent avoir le même niveau d'indentation.
6. Une instruction de retour est optionnelle. Le mot-clé **return** est utilisé pour quitter une fonction et pour retourner une valeur de la fonction.

```
def funct_nom(params):
    """docstring"""
    statement(s)
```

Fonctions

- **Sans return :**

```
def greet(name) :
    """
    Cette fonction salue la personne qui
    est passée en tant que un paramètre
    """
    print("Bonjour, " + name + "!")
```

  - **Console :**
  - **>>> greet('Paul')**
  - **>>> v = greet('Paul')**
  - **>>> print(v)**
  - **Sortie :**
  - **Bonjour, Paul!**
  - **None**

- **Avec return :**

```
def greet(name) :
    """
    Cette fonction salue la personne qui
    est passée en tant que un paramètre
    """
    return "Bonjour, " + name + "!"
```

  - **Console :**
  - **>>> greet('Paul')**
  - **>>> v = greet('Paul')**
  - **>>> print(v)**
  - **Sortie :**
  - **Bonjour, Paul!**
  - **Bonjour, Paul!**

Fonctions Récursives.

- **Récursivité :**
  - En Python, nous savons qu'une fonction peut appeler d'autres fonctions. Il est même possible que la fonction s'appelle elle-même. Ces types de construction sont appelés fonctions récursives.
  - La récursivité est un concept mathématique et de programmation. Elle signifie qu'une fonction s'appelle elle-même.
    - Cela a l'avantage de signifier que l'on peut faire une boucle à travers des données pour atteindre un résultat.

Fonctions Récursives.

```
def factoriel(x) :
    """Il s'agit d'une fonction récursive
    pour trouver la factorielle d'un entier """
    if x == 1 :
        return 1
    else :
        return (x * factoriel(x-1))
```

```
Console :

>>> num = 3
>>> print("La factorielle de", num, "est", factoriel(num))
```

Exécution :

factoriel(3) # 1er appel avec 3

3 * factoriel(2) # 2e appel avec 2

3 * 2 * factoriel(1) # 3ème appel avec 1

3 * 2 * 1 # retour du 3ème appel =1

3 * 2 # retour du 2e appel

6 # retour du 1er appel

Fonctions Récursives.

- **Notre récursivité se termine lorsque le nombre se réduit à une valeur. C'est ce qu'on appelle la condition de base/ condition d'arrêt.**
  - Toute fonction récursive doit avoir une condition de base qui arrête la récursivité, sinon la fonction s'appelle elle-même infiniment.
- **L'interpréteur Python limite les profondeurs de la récursivité pour éviter les récursivités infinies, ce qui entraîne des débordements de pile.**
  - Par défaut, la profondeur maximale de récursivité est de 1000. Si la limite est dépassée, il en résulte une RecursionError.

Exemple :

```
def recursive() :
    recursive()
```

Console :

```
>>>recursive()
```

Sortie :

```
Traceback (most recent call
last):
  File "<string>", line 3, in
<module>
  File "<string>", line 2, in a
  File "<string>", line 2, in a
  File "<string>", line 2, in a
  [Previous line repeated 996
more times]
RecursionError: maximum recursion
depth exceeded
```

Fonctions Récursives.

- **Avantages de la récursivité**
  - Les fonctions récursives donnent au code un aspect propre et élégant.
  - Une tâche complexe peut être décomposée en sous-problèmes plus simples grâce à la récursivité.
- **Inconvénients de la récursivité**
  - Parfois, la logique qui se trouve derrière la récursivité est difficile à suivre.
  - Les appels récursifs sont coûteux (inefficaces) car ils prennent beaucoup de mémoire et de temps.
  - Les fonctions récursives sont difficiles à déboguer.

Gestion des exceptions.

- **Les erreurs qui se produisent au moment de l'exécution (après avoir passé le test de syntaxe) sont appelées exceptions ou erreurs logiques.**
  - Par exemple, elles se produisent lorsque nous essayons d'ouvrir un fichier (pour le lire) qui n'existe pas (FileNotFoundError), d'essayer de diviser un nombre par zéro (ZeroDivisionError), ou d'essayer d'importer un module qui n'existe pas (ImportError).
- **Chaque fois que ce type d'erreur d'exécution se produit, Python crée un objet d'exception.**
- **Si l'exception n'est pas traité correctement, l'interpréteur imprime un retour à cette erreur avec quelques détails sur la raison pour laquelle cette erreur s'est produite.**

```
>>> 1/0


Traceback (most recent call last):

  File "<ipython-input-30-9e1622b385b6>", line
1, in <module>
    1/0

ZeroDivisionError: division by zero
```

```
>>> open("testNotFound.txt")


Traceback (most recent call last):

  File "<ipython-input-29-041f2d099717>", line
1, in <module>
    open("testNotFound.txt")

FileNotFoundError: [Errno 2] No such file or
directory: 'testNotFound.txt'
```

Gestion des exceptions.

- **Exceptions:**
  - Python a de nombreuses exceptions intégrées qui sont déclenchées lorsque votre programme rencontre une erreur.
  - Lorsque ces exceptions se produisent, l'interpréteur Python arrête le processus en cours et le transmet au processus appelant jusqu'à ce qu'il soit traité. S'il n'est pas traité, le programme se bloque.
  - En Python, les exceptions peuvent être traitées à l'aide d'une instruction **try**.

Gestion des exceptions.

- **L'opération critique qui peut soulever une exception est placée à l'intérieur de la clause try. Le code qui gère les exceptions est écrit dans la clause except.**
  - On peut donc choisir les opérations à effectuer une fois l'exception capturée.

- **Si aucune exception ne se produit, le bloc except est ignoré et le processus normal continue (pour la dernière valeur). Mais si une exception se produit, elle est prise par le bloc except (première et deuxième valeurs dans l'exemple).**

```
# import module sys to get the type of exception
import sys
randomList = ['a', 0, 2]
for entry in randomList:
    try:
        print("The entry is", entry)
        r = 1/int(entry)
        break
    except:
        print("Oops!", sys.exc_info()[0],
"occurred.")
        print("Next entry.")
        print()
print("The reciprocal of", entry, "is", r)
```

```
Out :
The entry is a
Oops! <class 'ValueError'> occurred.
Next entry.
The entry is 0
Oops! <class 'ZeroDivisionError'> occured.
Next entry.
The entry is 2
The reciprocal of 2 is 0.5
```

Gestion des exceptions.

- **Les exceptions spécifiques à Python**
  - Dans l'exemple ci-dessus, nous n'avons mentionné aucune exception spécifique dans la clause d'exception.
  - Ce n'est pas une bonne pratique de programmation, car elle permettra de prendre en compte toutes les exceptions et de traiter chaque cas de la même manière.
  - Une clause **try** peut comporter un nombre quelconque de clauses d'exception pour traiter différentes exceptions, mais une seule sera exécutée en cas d'exception.
  - Nous pouvons utiliser un ensemble de valeurs pour spécifier plusieurs exceptions dans une clause d'exception **except**.

```
try:
    # do something
    pass

except ValueError:
    # handle ValueError exception
    pass

except (TypeError, ZeroDivisionError):
    # handle multiple exceptions
    # TypeError and ZeroDivisionError
    pass

except:
    # handle all other exceptions
    pass
```

Gestion des exceptions.

- **Python try...finally**
  - La déclaration « **try** » en Python peut avoir une clause « **finally** » optionnelle. Cette clause est exécutée quoi qu'il arrive, et est généralement utilisée pour libérer des ressources externes.
    - Par exemple, nous pouvons être connectés à un centre de données distant via le réseau ou travailler avec un fichier ou une interface utilisateur graphique (GUI).
    - Nous devons nettoyer la ressource avant que le programme ne s'arrête, qu'il ait été exécuté avec succès ou non. Ces actions (fermeture d'un fichier, interface graphique ou déconnexion du réseau) sont effectuées dans la clause « **finally** » pour garantir l'exécution.

```
try:
   f = open("test.txt",encoding =
'utf-8')
   # perform file operations
finally:
   f.close()
```

Gestion des répertoires et des fichiers en Python

- **Fichiers :**
  - Du fait que la mémoire vive (RAM) est volatile (elle perd ses données lorsque l'ordinateur est éteint), nous utilisons les fichiers pour une utilisation future des données en les stockant de manière permanente.
  - Ainsi, en Python, une opération sur un fichier se déroule dans l'ordre suivant :
    - Ouvrir un fichier
    - Lire ou écrire (effectuer une opération)
    - Fermer le fichier

Gestion des répertoires et des fichiers en Python

- **Ouvrir un fichier :**
  - Python a une fonction **open()** intégrée pour ouvrir un fichier. Cette fonction renvoie un objet fichier, utilisé pour lire ou modifier le fichier

```
>>> f = open("test2.txt") # dans le répertoire courant
>>> f = open("C:/Python38/README.txt") # le chemin complet
```

  - Nous pouvons préciser le mode lors de l'ouverture d'un fichier.
    - Pour lire : mode = 'r', pour écrire : mode = 'w' et pour ajouter au fichier : mode = 'a'.
      - `f = open("test.txt", mode='w')`

Gestion des répertoires et des fichiers en Python

- **Fermer un fichier :**
  - Lorsque nous avons fini d'effectuer des opérations sur le fichier, nous devons le fermer correctement.
    - La fermeture d'un fichier libérera les ressources qui y étaient liées. Cela se fait à l'aide de la méthode close() en Python.

```
f = open("test2.txt")
# perform file operations
f.close()
```

  - Cette méthode n'est pas entièrement fiable.
    - Si une exception se produit lorsque nous effectuons une opération quelconque avec le fichier, le code sort sans fermer le fichier.
  - Une méthode plus sûre est d'utiliser un bloc **try...finally** qui assure que le fichier est fermé même si une exception se produit pendant l'exécution du programme.

```
try:
   f = open("test.txt")
   # perform file operations
finally:
   f.close()
```

Gestion des répertoires et des fichiers en Python

- **La solution la plus simple :**
  - La meilleure façon de fermer un fichier est d'utiliser la déclaration « **with** ». Cela garantit que le fichier est fermé lorsque le bloc à l'intérieur de la déclaration « **with** » est terminé.

```
with open("test.txt") as f:
   # perform file operations
```

  - Nous n'avons pas besoin d'appeler explicitement la méthode **close()**. Elle est faite en interne.

Gestion des répertoires et des fichiers en Python

- **Ecrire dans des fichiers:**
  - Pour pouvoir écrire dans un fichier en Python, il faut l'ouvrir en mode d'écriture **‘w’** ou en mode d’ajout **‘a’**.

```
with open("test.txt",'w') as f:
```

  - L'écriture dans le fichier se fait à l'aide de la méthode **write()**.

```
    f.write("my first file\n")
    f.write("This file\n\n")
    f.write("contains three lines\n")
```

Nous devons être prudents avec le mode ‘w’, car il écrasera le fichier s'il existe déjà. De ce fait, toutes les données précédentes sont effacées.

Gestion des répertoires et des fichiers en Python

- **Lecture de fichiers :**
  - Pour lire un fichier en Python, il faut ouvrir le fichier en mode de lecture « **r** » .
  - Différentes méthodes sont disponibles.
    - Nous pouvons utiliser la méthode **read(size)** pour lire le nombre de taille des données.
      - Si le paramètre de taille n'est pas spécifié, il lit et retourne jusqu'à la fin du fichier.
    -

```
>>> f = open("test.txt",'r')
```

```
>>> f.read(4)    # read the first 4 data
Out : 'This'
```

```
>>> f.read(4)    # read the next 4 data
Out : ' is '
```

```
>>> f.read()     # read in the rest till end of file
Out : 'my first file\nThis file\ncontains three lines\n'
```

```
>>> f.read()  # further reading returns empty sting
Out : ''
```

Gestion des répertoires et des fichiers en Python

- **Position du curseur**
  - Nous pouvons modifier le curseur (position) de notre fichier actuel en utilisant la méthode **seek()**.
  - De même, la méthode **tell()** renvoie notre position actuelle (en nombre d'octets).

```
>>> f.tell()    # get the current
file position
Out : 56
```

```
>>> f.seek(0)   # bring file cursor
to initial position
Out : 0
```

```
>>> print(f.read())  # read the
entire file
Out : This is my first file
    This file
    contains three lines
```

Gestion des répertoires et des fichiers en Python

- On peut lire un fichier ligne par ligne en utilisant une boucle *for*.

```
>>> for line in f:
...     print(line, end = '')
...
```

- nous pouvons utiliser la méthode **readline()** pour lire les lignes individuelles d'un fichier.

```
Out : This is my first file
      This file
      contains three lines
>>> f.readline()
Out : 'This is my first file\n'

>>> f.readline()
Out : 'This file\n'
```

- la méthode **readlines()** renvoie une liste des lignes restantes de l'ensemble du fichier. Toutes ces méthodes de lecture renvoient des valeurs vides lorsque la fin du fichier (EOF) est atteinte.

```
>>> f.readlines()
Out : ['This is my first file\n',
'This file\n', 'contains three
lines\n']
```

Gestion des répertoires et des fichiers en Python

- **Répertoire Python**

  - S'il y a un grand nombre de fichiers à traiter dans notre programme Python, nous pouvons organiser notre code dans différents répertoires pour rendre les choses plus gérables.

  - Un répertoire ou un dossier est un ensemble de fichiers et de sous-répertoires. Python dispose du module **os** qui nous fournit de nombreuses méthodes utiles pour travailler avec les répertoires (et les fichiers aussi).

Gestion des répertoires et des fichiers en Python

- **Obtenir un répertoire courant**
  - Nous pouvons obtenir le répertoire de travail actuel en utilisant la méthode **getcwd()** du module **os**.
  - Cette méthode renvoie le répertoire de travail actuel sous la forme d'une chaîne de caractères.

```
>>> import os
>>> os.getcwd()

Out[20]:
'/home/harmony1/Bureau/python
l1/code files'
```

- **Changement de répertoire**
  - Nous pouvons changer le répertoire de travail actuel en utilisant la méthode **chdir()**.
  - Le nouveau chemin que nous voulons changer doit être fourni comme une chaîne de caractères à cette méthode.

```
>>>
os.chdir('/home/harmony1/Burea
u/python l1')

>>>print(os.getcwd())

Out :
/home/harmony1/Bureau/python
l1
```

Gestion des répertoires et des fichiers en Python

- **Liste des répertoires et des fichiers**
  - Tous les fichiers et sous-répertoires à l'intérieur d'un répertoire peuvent être récupérés en utilisant la méthode **listdir()**.
  - Cette méthode prend un chemin et renvoie une liste des sous-répertoires et des fichiers dans ce chemin.
    - Si aucun chemin n'est spécifié, elle renvoie la liste des sous-répertoires et des fichiers du répertoire de travail courant.

```
>>> os.listdir("/")

Out[28]:
['proc',
 'dev',
 'sys',
 'initrd.img',
 'tmp',...]
```

Gestion des répertoires et des fichiers en Python

- **Créer un nouveau répertoire**
  - Nous pouvons créer un nouveau répertoire en utilisant la méthode **mkdir()**.
  - Cette méthode prend en compte le chemin du nouveau répertoire.
    - Si le chemin complet n'est pas spécifié, le nouveau répertoire est créé dans le répertoire actuel.

```
>>> os.mkdir('test')
>>> os.listdir()
['test']
```

- **Renommer un répertoire ou un fichier**
  - La méthode **rename()** permet de renommer un répertoire ou un fichier.
  - Pour renommer un répertoire ou un fichier, la méthode **rename()** prend deux arguments de base : le nom du fichier comme premier argument et le nouveau nom comme second argument.

```
>>> os.listdir()
['test']
>>> os.rename('test','new_one')
>>> os.listdir()
['new_one']
```

Gestion des répertoires et des fichiers en Python

- **Suppression d'un répertoire ou d'un fichier**
  - Un fichier peut être supprimé (effacé) en utilisant la méthode **remove()**.
  - De même, la méthode **rmdir()** supprime un répertoire vide.
  - Afin de supprimer un répertoire non vide, nous pouvons utiliser la méthode **rmtree()** à l'intérieur du module **shutil**.

```
>>> os.listdir()
['new_one', 'old.txt']
>>> os.remove('old.txt')
>>> os.listdir()
['new_one']
>>> os.rmdir('new_one')
>>> os.listdir()
[]
```

```
>>> os.listdir()
['test']

>>> os.rmdir('test')
Traceback (most recent call last):
...
OSError: [WinError 145] The directory is
not empty: 'test'

>>> import shutil
>>> shutil.rmtree('test')
>>> os.listdir()
[]
```

Fonction Lambda.

- **Que sont les fonctions lambda en Python ?**
  - Une fonction **lambda** est une petite fonction anonyme (définie sans nom).
    - Alors que les fonctions normales sont définies par le mot-clé **def** en Python, les fonctions anonymes sont définies par le mot-clé **lambda**.

Fonction Lambda.

- **Comment utiliser les fonctions lambda en Python ?**
  - **Syntaxe de la fonction Lambda en python :**

```
lambda arguments : expression
```

  - **Les fonctions lambda peuvent avoir un nombre quelconque d'arguments mais une seule expression.**
  - **L'expression est évaluée et renvoyée.**

On remplace la fonction :

```
def puissance(x,y):
    return x**y
```

Par :

```
puissance = lambda x,y : x ** y
```

Console :

```
>>> print(puissance(2,3))
```

Sortie :

```
Out : 8
```

Fonction Lambda.

- **Pourquoi utilise t-on les fonctions lambda ?**
  - Nous utilisons les fonctions lambda lorsque nous avons besoin d'une fonction sans nom pendant une courte période.
  - Généralement, les fonctions **lambda** comme argument d'une fonction d'ordre supérieur (une fonction qui prend d'autres fonctions comme arguments) comme filter(), map() etc.

Fonctions Map.

- **La fonction map() :**
  - La fonction **map()** en Python prend en entrée une fonction et une liste.
  - La fonction est appelée avec tous les éléments de la liste et une nouvelle liste est renvoyée qui contient les éléments renvoyés par cette fonction pour chaque élément.

Exemple :

```
liste = [1, 5, 4, 6, 8, 11, 3, 12]

new_liste =
list(map(lambda x : x * 2 , liste))
```

Console :

```
>>> print(new_liste)
```

Sortie :

```
Out : [2, 10, 8, 12, 16, 22, 6, 24]
```

Fonction Filter.

- **La fonction filter() :**
  - En Python, la fonction **filter()** prend en argument une fonction et une liste.
  - La fonction est appelée avec tous les éléments de la liste et une nouvelle liste est renvoyée qui contient les éléments pour lesquels la fonction évalue à True.

Exemple :

```
my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(filter(lambda x: (x%2 == 0) , my_list))
```

Console :

```
>>> print(new_list)
```

Sortie :

```
Out : [4,6,8,12]
```

Listes en compréhension.

- **Liste en compréhension avec boucle**

boucle :

```
letters = []
for letter in 'human':
    letters.append(letter)
```

Liste en compréhension :

```
letters = [ letter for letter in 'human' ]
```

Console :

```
 >>> print(letters)
```

Sortie:

```
Out : ['h', 'u', 'm', 'a', 'n']
```

- **La liste en compréhension est une façon élégante de définir et de créer des listes basées sur des listes existantes.**
  - **Syntaxe de la liste en compréhension**
    - [expression for un élément in liste]
    - [i**2 for i in liste]

- **Dans l'exemple, une nouvelle liste est assignée à la variable letters, et list contient les éléments de la chaîne de caractères itérable "human".**

Listes en compréhension.

- **Liste de compréhension vs Lambda**
  - La **liste en compréhension** n'est pas la seule façon de travailler sur les listes. Diverses fonctions intégrées et fonctions lambda permettent de créer et de modifier des listes en moins de lignes de code.
  - Cependant, les listes sont généralement plus lisibles que les fonctions lambda. Il est plus facile de comprendre ce que le programmeur essayait d'accomplir lorsque la **liste en compréhension** est utilisée.

Lambda :

```
letters = list(map(lambda x: x, 'human'))
```

Liste en compréhension :

```
letters = [ letter for letter in 'human' ]
```

Console :

```
 >>> print(letters)
```

Sortie:

```
Out : ['h', 'u', 'm', 'a', 'n']
```

Listes en compréhension.

- **Liste de compréhension avec conditions**

if :

```
letters = [ letter for letter in 'human'
if letter == ‘h’ or letter == ’u’ ]
```

if… else :

```
letters = [ letter if letter ==’h’ else
‘h’ for letter in 'human' ]
```

Console :

```
 >>> print(letters)
```

Sortie:

```
Out1 : ['h', 'u']

Out2 : ['h', 'h', 'h', 'h', 'h']
```

- **La liste en compréhension peut utiliser la déclaration conditionnelle pour modifier une liste existante.**
- **Syntaxe de la liste de compréhension avec des conditions :**
  - [expression if condition else expression2 for élément in liste]
  - [expression for élément in liste if condition1]

Listes en compréhension.

- **Liste en compréhension avec boucles imbriquées**

- **Dans l’exemple, pour i dans l'intervalle(2) est exécuté avant la ligne [i] pour la ligne dans la matrice. Ainsi, au début, une valeur est attribuée à i, puis l'élément dirigé par la rangée [i] est ajouté dans la variable de transposition.**

Boucles imbriquées :

```
transposed = []
matrix = [[1, 2], [3, 4], [4, 5], [6, 8]]

for i in range(len(matrix[0])):
    transposed_row = []
    for row in matrix:
        transposed_row.append(row[i])
    transposed.append(transposed_row)
```

Liste en compréhension :

```
transposed = [[row[i] for row in matrix] for i in range(2)]
```

Console :

```
>>> print(transposed)
```

Sortie:

```
Out: [[1, 3, 4, 6], [2, 4, 5, 8]]
```

Projet

- **Il s'agit d'un projet noté qui vise à vous faire utiliser les connaissances que vous avez acquises pour pratiquer et maîtriser les fonctionnalités python dont vous aurez besoin pour le suite de votre parcours.**
- **Ce projet comporte quatre parties principales :**
  - la première consiste à réaliser le jeu du Pendu.
  - La deuxième partie consiste à coder des fonctions récursives.
  - La troisième partie consiste à utiliser des fonctions python avancées telles que lambda, map, filter et listes de compréhension.
  - Enfin, la quatrième partie consiste à combiner les trois parties (importer les modules qui correspondent à chaque partie) et implémenter les fonctions dans un seul fichier principal qui permettra à l'utilisateur d'utiliser votre programme.
- **Il est obligatoire de commenter votre code et d'expliquer en détail ce que vous faites dans chacune de vos fonctions ou scripts !**

- Lien du cours+projet :

  https://github.com/nassib/python-l1-2020-2021

- E-mail:

  Nassib.abdallah@univ-angers.fr

À vous de jouer…