Course: Python

Ali ZAINOUL <contact@clearcode.fr>

Crystal Clear Code
June 16, 2025

- Les bonnes pratiques

- Paquets
  - Importation de paquet
  - Création d'un paquet
  - Création d'un paquet (__init__.py)
- Tests unitaires : instruction `assert`, module `unittest`
  - Framework `pytest`

** Modules de la bibliothèque standard **

Interaction avec l'interpréteur : module `sys`

- Le module `sys` fournit des fonctionnalités permettant d'interagir avec l'interpréteur Python.
- Il permet d'accéder à des informations sur l'environnement d'exécution, de manipuler le chemin de recherche des modules, et plus encore.

Exemple d'utilisation

```python
import sys

# Affichage de la version de Python
print("Python version:", sys.version)

# Affichage du chemin de recherche des modules
print("Module search path:", sys.path)
```

Interaction avec le système d'exploitation : modules `os` et `pathlib`

- Les modules `os` et `pathlib` fournissent des fonctionnalités pour interagir avec le système d'exploitation et manipuler les chemins de fichiers de manière efficace.
- Le module `os` permet d'exécuter des opérations système telles que la création de répertoires, la navigation dans le système de fichiers, etc.
- Le module `pathlib` fournit une interface orientée objet pour manipuler les chemins de fichiers et de répertoires de manière portable.

Exemple d'utilisation

```python
import os
from pathlib import Path

# Création d'un répertoire
os.makedirs("mydir")

# Récupération du répertoire courant
current_dir = Path.cwd()
print("Current directory:", current_dir)

# Vérification de l'existence d'un fichier
file_path = Path("myfile.txt")
if file_path.exists():
    print("File exists!")
else:
    print("File does not exist.")
```

Interaction avec le système de fichiers : module `os.path`

- Le module `os.path` fournit des fonctionnalités pour manipuler les chemins de fichiers de manière efficace.
- Il permet de vérifier l'existence de fichiers, de répertoires, de manipuler les extensions de fichier, etc.

Exemple d'utilisation

```python
import os.path

# Vérifier l'existence d'un fichier
file_path = "example.txt"
if os.path.exists(file_path):
    print("Le fichier existe:", file_path)
else:
    print("Le fichier n'existe pas:", file_path)

# Obtenir le répertoire parent d'un fichier
parent_directory = os.path.dirname(file_path)
print("Répertoire parent:", parent_directory)

# Obtenir l'extension d'un fichier
file_extension = os.path.splitext(file_path)[1]
print("Extension de fichier:", file_extension)
```

Expressions rationnelles : module `re`

- Le module `re` permet de travailler avec des expressions rationnelles en Python.
- Il fournit des fonctionnalités pour rechercher, extraire et manipuler des motifs de texte basés sur des modèles définis.

Module `datetime`

- Le module `datetime` fournit des classes pour manipuler des dates et des heures en Python.
- Il permet de créer, manipuler et formater des objets de date et d'heure.

Exemple d'utilisation

```python
from datetime import datetime

# Création d'un objet de date et d'heure
now = datetime.now()
print("Date and time:", now)

# Formatage de la date et de l'heure
formatted_date = now.strftime("%Y-%m-%d %H:%M:%S")
print("Formatted date and time:", formatted_date)

# Extraction des composants de la date et de l'heure
year = now.year
month = now.month
day = now.day
hour = now.hour
minute = now.minute
second = now.second
print("Year:", year)
print("Month:", month)
print("Day:", day)
print("Hour:", hour)
print("Minute:", minute)
print("Second:", second)
```

Module `math`

- Le module `math` fournit des fonctions mathématiques standard en Python.
- Il inclut des fonctions trigonométriques, logarithmiques, de puissance, etc.

Exemple d'utilisation

```python
import math

# Exemples de fonctions mathématiques
print("Sin(30 degrees):", math.sin(math.radians(30)))  # Output: 0.49999999999999994
print("Cos(45 degrees):", math.cos(math.radians(45)))  # Output: 0.7071067811865476
print("Tan(60 degrees):", math.tan(math.radians(60)))  # Output: 1.7320508075688767
print("Logarithm base 2 of 16:", math.log2(16))          # Output: 4.0
print("Square root of 25:", math.sqrt(25))               # Output: 5.0
print("Factorial of 5:", math.factorial(5))              # Output: 120
```

Module `timeit`

- Le module `timeit` est utilisé pour mesurer le temps d'exécution de petits fragments de code Python.
- Il fournit un moyen simple de comparer les performances de différentes implémentations.

Exemple d'utilisation

```python
import timeit

# Exemple de mesure du temps d'exécution
time_taken = timeit.timeit(stmt='[i**2 for i in range(1000)]', number=10000)
print("Time taken:", time_taken)
```

Module `urllib`

- Le module `urllib` est utilisé pour récupérer des données à partir d'URLs en Python.
- Il fournit des fonctionnalités pour ouvrir, lire et analyser des pages web.

Exemple d'utilisation

```python
import urllib.request

# Exemple de récupération de données à partir d'une URL
response = urllib.request.urlopen('https://www.example.com')
html = response.read()
print(html)
```

Module `collections`

- Le module `collections` fournit des classes spécialisées de conteneurs de données en Python.
- Il inclut des classes telles que `Counter`, `deque`, `namedtuple`, etc., qui étendent les fonctionnalités des types de données intégrés.

Exemple d'utilisation

```python
import collections

# Exemple d'utilisation de Counter
word_counts = collections.Counter(['apple', 'banana', 'apple', 'orange', 'banana'])
print("Word counts:", word_counts)
```

Module `csv`

- Le module `csv` est utilisé pour lire et écrire des fichiers CSV (Comma-Separated Values) en Python.
- Il fournit des fonctionnalités pour travailler avec des données tabulaires stockées dans des fichiers CSV.

Exemple d'utilisation

```python
import csv

# Exemple de lecture d'un fichier CSV
with open('data.csv', 'r') as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

Module `json`

- Le module `json` est utilisé pour travailler avec des données JSON (JavaScript Object Notation) en Python.
- Il fournit des fonctionnalités pour sérialiser et désérialiser des objets Python en JSON et vice versa.

Exemple d'utilisation

```python
import json

# Exemple de sérialisation et désérialisation JSON
data = {'name': 'John', 'age': 30}
json_string = json.dumps(data)
print("JSON string:", json_string)

parsed_data = json.loads(json_string)
print("Parsed data:", parsed_data)
```

Module `sqlite3`

- Le module `sqlite3` est utilisé pour travailler avec des bases de données SQLite en Python.
- Il fournit des fonctionnalités pour exécuter des requêtes SQL, récupérer des données et gérer les transactions.

Exemple d'utilisation

```python
import sqlite3

# Exemple de création d'une base de données SQLite
conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Création d'une table
cursor.execute('''CREATE TABLE IF NOT EXISTS stocks
                  (date text, trans text, symbol text, qty real, price real)''')

# Insertion de données
cursor.execute("INSERT INTO stocks VALUES ('2024-04-30', 'BUY', 'GOOG', 100, 1225.12)")

# Sauvegarde des modifications
conn.commit()

# Fermeture de la connexion
conn.close()
```

** Les modules et les paquets **

Modules en Python : Introduction (1/3)

- **Qu'est-ce que les modules en Python ?** En Python, un module est un fichier contenant du code Python qui peut être importé dans d'autres scripts Python. Les modules sont utilisés pour organiser le code dans des fichiers distincts et fournissent un moyen de réutiliser le code dans différents programmes. Pour créer un module, il suffit d'écrire du code Python dans un fichier avec une extension `.py`. Vous pouvez ensuite importer le module dans un autre script Python à l'aide de l'instruction **import**.

Modules en Python : Importation (2/3)

- **Comment importer des modules en Python ?** Pour importer un module en Python, utilisez l'instruction **import** suivie du nom du module. Par exemple, si vous avez un module nommé `my_module.py`, vous pouvez l'importer dans un autre script Python avec l'instruction suivante :

```
1    import my_module
```

Modules en Python : Utilisation (3/3)

- Une fois que vous avez importé le module, vous pouvez utiliser ses fonctions et variables dans votre code. Pour appeler une fonction du module, utilisez la notation pointée, comme ceci :

```
1    my_module.my_function()
```

- Vous pouvez également utiliser le mot-clé **from** pour importer des classes, des fonctions ou des variables spécifiques d'un module, comme ceci :

```
1    from my_module import my_class, my_function, my_variable
```

- Cela vous permet d'utiliser les classes, les fonctions et les variables importées sans avoir à les préfixer par le nom du module.

Bloc `if __name__ == "__main__"`

- En Python, le bloc `if __name__ == "__main__"` est utilisé pour déterminer si le fichier est exécuté en tant que script principal ou s'il est importé en tant que module dans un autre script.
- Le code à l'intérieur du bloc `if __name__ == "__main__"` ne sera exécuté que si le fichier est exécuté en tant que script principal.
- Cela empêche l'exécution non voulue du code lorsque le fichier est importé en tant que module.

Bloc `if __name__ == "__main__"` - Exemple

```python
# module.py
def function():
    print("Function called")

if __name__ == "__main__":
    # Code to execute only if this script is run as the main
       program
    print("Module is being run directly")
    function()
else:
    # Code to execute if this script is imported as a module
    print("Module is being imported")
```

Bonnes pratiques pour les modules en Python (1/2)

- Lors de l'importation de modules en Python, il existe certaines bonnes pratiques que vous devriez suivre pour vous assurer que votre code est propre, **lisible** et **maintenable**. Voici quelques conseils:

  - Utilisez des importations absolues pour spécifier le chemin complet du module que vous souhaitez importer. Cela permet de préciser le module que vous importez et aide à éviter les conflits de noms.
  - Évitez d'utiliser des importations par joker (par exemple: `from mon_module import *`) car elles peuvent rendre difficile la compréhension d'où proviennent les fonctions et variables.
  - Utilisez des alias pour raccourcir les noms de module et les rendre plus faciles à utiliser. Par exemple, vous pourriez importer le module `numpy` de cette manière : `import numpy as np`.

Bonnes pratiques pour les modules en Python (2/2)

- Suite :
  - Regroupez les importations liées ensemble en haut de votre script pour indiquer clairement les modules dont dépend votre code.
  - Évitez les importations circulaires, où deux modules dépendent l'un de l'autre. Cela peut créer des dépendances confuses et rendre votre code plus difficile à comprendre.
- En suivant ces bonnes pratiques, vous pouvez vous assurer que votre code Python est bien organisé, facile à lire et maintenable au fil du temps.
- Bonnes pratiques.

Importation de paquet

- En Python, un paquet est simplement un répertoire contenant un fichier spécial `__init__.py`.
- L'importation de paquet permet d'organiser et de structurer le code en regroupant des modules connexes dans un même répertoire.
- L'importation de paquet se fait avec la syntaxe `import name_package.name_module`.
- Pour importer tous les modules d'un paquet, on utilise la syntaxe `from name_package import *`, mais cela est généralement déconseillé pour éviter les conflits de noms.

Importation de paquet - Exemple

```python
# Paquet: mypackage
# Structure:
# mypackage/
#       __init__.py
#       module1.py
#       module2.py

# Content of file __init__.py
# (may be empty or containing initializations)
# __init__.py

# Importing a module from a package
import mypackage.module1
# Using the module
mypackage.module1.function()

# Importing of all modules from a package
from mypackage import *
# Utilisation des modules importés
module1.function()
module2.function()

# Importing using alias
import mypackage.module1 as m1
# Using with alias
m1.function()
```

Création de paquet

- En Python, un paquet est simplement un répertoire contenant un fichier spécial `__init__.py`.
- Le fichier `__init__.py` peut être vide ou contenir des initialisations.
- La création de paquet permet d'organiser et de structurer le code en regroupant des modules connexes dans un même répertoire.
- Pour créer un paquet, il suffit de créer un répertoire et d'y ajouter un fichier `__init__.py`.

Création de paquet - Exemple

```
# Paquet: mypackage
# Structure tree:
# mypackage/
#       __init__.py
#       module1.py
#       module2.py

# Content of file __init__.py
# (may be empty or containing initializations)
# __init__.py

# Content of module1.py
# module1.py
def function1():
    print("Function 1 in module 1")

# Content of module2.py
# module2.py
def function2():
    print("Function 2 in module 2")
```

Création d'un paquet

- Pour créer un paquet en Python, il suffit de créer un répertoire avec un fichier `__init__.py` à l'intérieur.
- Le fichier `__init__.py` peut être vide ou contenir du code d'initialisation pour le paquet.
- Voici un exemple de structure de répertoire pour un paquet nommé `mypackage`:

```
mypackage/
    __init__.py
    module1.py
    module2.py
```

- Dans cet exemple, `mypackage` est un paquet Python contenant deux modules, `module1` et `module2`.

Example 1: Creating a Package - Structure

```
mypackage/
   __init__.py
   module1.py
   module2.py
```

Example 1: Package Initialization (__init__.py)

```python
# __init__.py

from .module1 import MyClass1
from .module2 import MyClass2
```

Example 1: Module 1 (module1.py)

```python
# module1.py

class MyClass1:
    def __init__(self):
        print("Initializing MyClass1")

    def method1(self):
        print("Method 1 in MyClass1")
```

Example 1: Module 2 (module2.py)

```python
# module2.py

class MyClass2:
    def __init__(self):
        print("Initializing MyClass2")

    def method2(self):
        print("Method 2 in MyClass2")
```

Example 2: Creating a Package - Structure

```
mypackage/
   __init__.py
   utils/
       __init__.py
       helper1.py
       helper2.py
   main.py
```

Example 2: Package Initialization (__init__.py)

```python
# __init__.py

from .utils.helper1 import Helper1
from .utils.helper2 import Helper2
```

Example 2: Module 1 (helper1.py)

```python
# helper1.py

class Helper1:
    def __init__(self):
        print("Initializing Helper1")

    def method1(self):
        print("Method 1 in Helper1")
```

Example 2: Module 2 (helper2.py)

```python
# helper2.py

class Helper2:
    def __init__(self):
        print("Initializing Helper2")

    def method2(self):
        print("Method 2 in Helper2")
```

Example 2: Main Module (main.py)

```python
# main.py

from mypackage.utils import Helper1, Helper2

helper1 = Helper1()
helper1.method1()

helper2 = Helper2()
helper2.method2()
```

** Tests unitaires **

Tests unitaires : instruction `assert`, module `unittest`

- L'instruction `assert` est utilisée pour vérifier si une expression est vraie. Si l'expression est fausse, une erreur AssertionError est levée.
- Le module `unittest` fournit un framework pour écrire et exécuter des tests unitaires en Python.

Exemple d'utilisation

```python
import unittest

def add(a, b):
    return a + b

class TestAddFunction(unittest.TestCase):

    def test_add_positive_numbers(self):
        self.assertEqual(add(1, 2), 3)

    def test_add_negative_numbers(self):
        self.assertEqual(add(-1, -2), -3)

    def test_add_mixed_numbers(self):
        self.assertEqual(add(1, -2), -1)
        self.assertEqual(add(-1, 2), 1)

if __name__ == "__main__":
    unittest.main()
```

Framwork `pytest`

- `pytest` est un framework de test populaire et flexible pour Python.
- Il permet d'écrire des tests simples et complexes de manière concise.
- `pytest` offre des fonctionnalités avancées telles que la découverte automatique des tests, la gestion des fixtures et des rapports détaillés.
- Il supporte également les tests de performance et les tests de manière asynchrone.

Exemple d'utilisation

```python
# Exemple de test avec pytest

def test_upper():
    assert 'hello'.upper() == 'HELLO'

def test_isupper():
    assert 'HELLO'.isupper()
    assert not 'Hello'.isupper()
```

- Il suffit de naviguer dans le dossier où se trouve le fichier des tests unitaires exemple: `test_myfile.py` et de lancer la commande: `pytest` afin de lancer les tests unitaires.

Utilisation des Fixtures

- Les fixtures permettent de configurer un état initial avant d'exécuter des tests.
- Elles peuvent être partagées entre plusieurs tests, ce qui évite la répétition de code.

```python
import pytest

@pytest.fixture
def sample_data():
    return [1, 2, 3]

def test_sum(sample_data):
    assert sum(sample_data) == 6
```