Chapitre 1 : Rappels sur la programmation en Python

Dr. Ishak ABDI — ishak.abdi@univ-jijel.dz
Département de Génie civil et hydraulique — Université de Jijel

Boucles For imbriquées (Nested For Loops)

Principe

Une boucle peut être placée **à l'intérieur d'une autre boucle**.
Cela permet, par exemple, de parcourir des tableaux 2D, créer des matrices ou effectuer des répétitions multiples.

Syntaxe générale

```
for ii in range(...):
    for jj in range(...):
        # instructions
```

Exercice — Nombres premiers

- Les **25 premiers nombres premiers** (tous les nombres premiers inférieurs à 100) sont : `2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97`
- Par définition, un nombre premier n'a que **1** et **lui-même** comme diviseurs.
- S'il possède un autre diviseur, alors il **n'est pas premier**.
- Un nombre naturel (1, 2, 3, 4, 5, 6, etc.) est appelé **nombre premier** s'il est **strictement supérieur à 1** et **ne peut pas être écrit comme un produit de deux nombres naturels plus petits que lui**.
- Créer un script Python qui trouve **tous les nombres premiers entre 1 et 200**.

solution

```
# Liste pour stocker les nombres premiers
nombres_premiers = []

# Boucle pour tester tous les nombres de 1 à 200
for nombre in range(2, 201):  # On commence à 2 car 1 n'est pas premier
    est_premier = True
    for i in range(2, int(nombre**0.5) + 1):
        if nombre % i == 0:
            est_premier = False
            break
    if est_premier:
        nombres_premiers.append(nombre)

# Affichage des nombres premiers
print("Nombres premiers entre 1 et 200 :")
print(nombres_premiers)
```

5.4 Boucles *while*

La boucle **while** répète un ensemble d'instructions un nombre indéfini de fois, tant qu'une condition logique est vraie.

Exemple — Utilisation des boucles *while* en Python

```
m = 8

while m > 2:
    print(m)
    m = m - 1
```

exemple suite de Fibonacci

```
limite = int(input("Entrez la limite maximale: "))

a, b = 0, 1
print("Suite de Fibonacci:")
print(a)

while b <= limite:
    print(b)
    a = b
    b = a + b
```

Les fonctions en Python

- Définition : bloc de code qui s'exécute lorsqu'il est appelé.
- Paramètres : données que l'on peut transmettre à la fonction.
- Retour : une fonction peut renvoyer un résultat.
- Nous avons déjà utilisé de nombreuses fonctions intégrées.
  `print()`, `len()`, `input()`, `type()`

Scripts vs Fonctions

- Il est important de connaître la différence entre un script et une fonction.

Scripts

- Collection de commandes que l'on exécute dans l'éditeur.
- Utilisés pour automatiser des tâches répétitives.

Fonctions

- Opèrent sur des informations (entrées) et renvoient des résultats (sorties).
- Possèdent un espace de travail séparé et des variables internes valides uniquement à l'intérieur de la fonction.

Fonctions définies par l'utilisateur

- Vos propres fonctions fonctionnent de la même manière que les fonctions intégrées que vous utilisez souvent, comme plot(), `rand()`, `mean()`, `std()`, etc.
- **Python** propose de nombreuses fonctions intégrées, mais il est souvent nécessaire de créer vos propres fonctions (appelées fonctions définies par l'utilisateur).
- En **Python**, une fonction est définie avec le mot-clé `def` :

```
def nom_de_la_fonction(paramètres):
    # corps de la fonction
    return résultat
```

Exemple – Créer une fonction dans un fichier séparé

- Créer un fichier Python séparé nommé `myfunctions.py`
- Définir la fonction dans ce fichier :

```
def average(x, y):
    return (x + y)
```

- Ensuite, nous créons un nouveau fichier Python (par exemple, `testaverage.py`) dans lequel nous utilisons la fonction que nous avons créée.

```
from myfunctions import average
a = 2
b = 3
c = average(a, b)
print(c)  # Affiche 5
```

Exemple – Créer une fonction avec plusieurs valeurs de retour

```
def stat(data):
    total_sum = 0

    # Calcul de la somme de tous les nombres
    for x in data:
        total_sum += x

    # Calcul de la moyenne
    N = len(data)
    mean = total_sum / N

    # Retourne la somme et la moyenne
    return total_sum, mean
```

```
data = [1 ,5 ,6 ,3 ,12, 3]

totalsum, mean = stat(data)

 print(totalsum ,mean)
```

Exercice 6.3.2 – Conversion radians ↔️ degrés

- La plupart des fonctions trigonométriques utilisent les **radians**.
- Créez vos propres fonctions pour **convertir entre radians et degrés**.
  - De radians à degrés : $d[^\circ] = r[\mathrm{rad}] \times \frac{180}{\pi}$
  - De degrés à radians : $r[\mathrm{rad}] = d[^\circ] \times \frac{\pi}{180}$
- Créer deux fonctions :
  - `r2d(x)` → convertit les **radians en degrés**
  - `d2r(x)` → convertit les **degrés en radians**
- Sauvegarder ces fonctions dans **un fichier Python** `.py`.

```python
import math

# Conversion radians → degrés
def r2d(x):
    return x * (180 / math.pi)

# Conversion degrés → radians
def d2r(x):
    return x * (math.pi / 180)
```

Création de classes en Python

- Python est un langage **orienté objet (OOP)**.
- Presque tout en Python est un **objet**, avec ses **propriétés** et **méthodes**.
- La base de la programmation orientée objet est la **classe**.
- Pour créer une classe, utilisez le mot-clé :

```
class NomDeLaClasse:
    # corps de la classe
```

Exemple – Classe simple

- Création d'une classe `Car` :

```
class Car:
    model = "Volvo"
    color = "Blue"

# Création d'un objet à partir de la classe
car = Car()

# Accès aux attributs de l'objet
print(car.model)  # Volvo
print(car.color)  # Blue
```

- `Car` est une classe avec deux attributs : `model` et `color`.
- `car` est un objet de la classe `Car`.
- On peut accéder aux attributs via objet.attribut.

7.2 La fonction `__init__()`

- En Python, toutes les classes ont une fonction intégrée `__init__()`.
- Elle est **toujours exécutée** lors de l'initialisation d'un objet.
- Dans beaucoup d'autres langages OOP, on l'appelle **constructeur**.

Exercice – Exemple avec `__init__()`

```
class Car:
    def __init__(self, model, color):
        self.model = model
        self.color = color

# Création d'un objet avec initialisation
car1 = Car("Ford", "Green")

# Accès aux attributs
print(car1.model)  # Ford
print(car1.color)  # Green
```

- self fait référence à l'objet courant.
- On peut initialiser les attributs dès la création de l'objet.

```python
# Définition de la classe Car avec une méthode
class Car:
    def __init__(self, model, color):
        self.model = model
        self.color = color

    def displayCar(self):
        print(self.model)
        print(self.color)

# Utilisation de la classe
# Objet 1
car1 = Car("Tesla", "Red")
car1.displayCar()
# Affiche:
# Tesla
# Red
# Objet 2
car2 = Car("Ford", "Green")
print(car2.model)  # Ford
print(car2.color)  # Green

# Objet 3
car3 = Car("Volvo", "Blue")
print(car3.model)  # Volvo
print(car3.color)  # Blue

# Modification de l'attribut
car3.color = "Black"
car3.displayCar()
# Affiche:
# Volvo
# Black
```

```python
# Définition de la classe Student
class Student:
    def __init__(self, name, grades=None):
        self.name = name
        if grades is None:
            self.grades = []
        else:
            self.grades = grades

    def add_grade(self, grade):
        if 0 <= grade <= 20:
            self.grades.append(grade)
            print(f"Note {grade} ajoutée pour {self.name}")
        else:
            print("Note invalide ! Elle doit être entre 0 et 20.")

    def average(self):
        if self.grades:
            return sum(self.grades) / len(self.grades)
        else:
            return 0

    def display_student(self):
        print(f"Étudiant : {self.name}")
        print(f"Notes : {self.grades}")
        print(f"Moyenne : {self.average():.2f}")

# Création de quelques étudiants
student1 = Student("Alice", [15, 18, 12])
student2 = Student("Bob")

# Utilisation des méthodes
student1.display_student()
student2.add_grade(14)
student2.add_grade(16)
student2.display_student()
```