Cours Python EPSI

Cours 2 : Programmation Orientée Objet Avancée, Exceptions, et Concepts Complémentaires

Objectifs

Dans ce cours, nous allons approfondir la Programmation Orientée Objet (POO) en abordant des concepts comme l'héritage, le polymorphisme, la gestion des exceptions, la création d'exceptions personnalisées, le pattern matching, le mot-clé `with`, les classes "auto-closeable, les générateurs, l'utilisation de `functools`, et la notion de complexité algorithmique (Big-O).

Héritage et Polymorphisme

Héritage

L'héritage permet à une classe d'hériter des attributs et méthodes d'une autre classe, appelée super-classe. Cela permet de réutiliser le code et de le spécialiser pour créer de nouvelles fonctionnalités.

Exemple d'héritage :

```python
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return f"{self.name} fait du bruit."


class Dog(Animal):
    def speak(self):
        return f"{self.name} aboie."


class Cat(Animal):
    def speak(self):
        return f"{self.name} miaule."
```

```
dog = Dog("Rex")
cat = Cat("Whiskers")

print(dog.speak())  # Rex aboie.
print(cat.speak())  # Whiskers miaule.
```

Polymorphisme

Le polymorphisme permet à des objets de classes différentes de répondre à la même méthode d'une manière spécifique à leur type. Cela rend le code plus flexible.

Exemple de polymorphisme :

```
def make_animal_speak(animal):
    print(animal.speak())


dog = Dog("Rex")
cat = Cat("Whiskers")

make_animal_speak(dog)  # Rex aboie.
make_animal_speak(cat)  # Whiskers miaule.
```

Note : Contrairement à d'autres langages (C++/Java), Python utilise la notion de "duck-typing"

> Duck typing is a principle used in dynamically typed programming languages like Python in which the code isn't constrained or bound to specific data types. Instead, it will execute an action depending on the type of built-in objects being processed.8 juil. 2024

Grossièrement, si ça a 2 pattes et un bec, c'est forcément un canard.

=> Si deux classes ont les mêmes méthodes, sans hériter l'une de l'autre, on peut malgré tout les stocker comme si elles héritaient d'une classe commune.

Gestion des Exceptions

Bloc `try/except`

Le bloc `try` capture les exceptions potentielles, et le bloc `except` gère les erreurs rencontrées.

Exemple :

```
try:
    x = 10 / 0
except ZeroDivisionError:
    print("Erreur : division par zéro.")
```

Bloc `finally`

Le bloc `finally` s'exécute toujours, que l'exception soit levée ou non, et est utilisé pour les opérations de nettoyage.

Exemple :

```
try:
    file = open("example.txt", "r")
finally:
    file.close()
```

Création d'Exceptions Personnalisées

Vous pouvez créer des exceptions personnalisées en héritant de la classe `Exception`. Cela permet de mieux gérer les erreurs spécifiques à votre application.

Exemple :

```
class InvalidAgeError(Exception):
    def __init__(self, age, message="L'âge doit être compris entre 0 et
    ↪  120."):
        self.age = age
        self.message = message
        super().__init__(self.message)

    def __str__(self):
        return f"{self.age} -> {self.message}"


def check_age(age):
    if not (0 <= age <= 120):
        raise InvalidAgeError(age)
    print(f"L'âge {age} est valide.")
```

```python
try:
    check_age(150)
except InvalidAgeError as e:
    print(e)
```

Pattern Matching (Python 3.10+)

Le pattern matching introduit dans Python 3.10 permet de faire correspondre des structures de données complexes avec des motifs spécifiques et de gérer chaque cas de manière distincte.

Exemple basique :

```python
def describe_point(point):
    match point:
        case (0, 0):
            return "Origine"
        case (0, y):
            return f"Axe Y, à {y}"
        case (x, 0):
            return f"Axe X, à {x}"
        case (x, y):
            return f"Point à ({x}, {y})"
        case _:
            return "C'est autre chose"


print(describe_point((0, 0)))  # Origine
```

Pattern Matching avec des Classes :

```python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y


def process_point(point):
    match point:
```

```
        case Point(x=0, y=0):
            return "Origine"
        case Point(x=0, y=y):
            return f"Sur l'axe Y à {y}"
        case Point(x=x, y=0):
            return f"Sur l'axe X à {x}"
        case Point(x=x, y=y):
            return f"Point à ({x}, {y})"


point = Point(0, 5)
print(process_point(point))  # Sur l'axe Y à 5
```

Mot-clé `with` et Classes Auto-Fermetures

Utilisation du mot-clé `with`

Le mot-clé `with` assure que les ressources sont libérées correctement après leur utilisation, même en cas d'exception.

Exemple :

```
with open('example.txt', 'r') as file:
    content = file.read()
    print(content)
```

Dans cet exemple, le fichier est automatiquement fermé après la fin du bloc `with`.

Création de Classes Auto-Fermetures

Pour créer des classes compatibles avec le mot-clé `with`, il faut implémenter les méthodes `__enter__` et `__exit__`.

Exemple :

```
class Resource:
    def __enter__(self):
        print("Acquisition de la ressource")
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        print("Libération de la ressource")
```

```
    def do_something(self):
        print("Utilisation de la ressource")


with Resource() as resource:
    resource.do_something()
```

Générateurs et `yield`

Les générateurs produisent des séquences de valeurs à l'aide du mot-clé `yield`, permettant une gestion plus efficace de la mémoire.

Exemple :

```
def count_up_to(max):
    count = 1
    while count <= max:
        yield count
        count += 1


for number in count_up_to(5):
    print(number)
```

Functools : Fonctions Utilitaires Avancées

La bibliothèque **`functools`** fournit des outils utiles pour la manipulation des fonctions, comme la mémorisation, la composition, et les appels partiels.

`functools.lru_cache` : Mémorisation

Le décorateur `lru_cache` permet de mémoriser les résultats de fonctions afin d'éviter de les recalculer.

Exemple :

```
import functools

@functools.lru_cache(maxsize=128)
```

```python
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(50))
```

`functools.partial` : Application Partielle

La fonction `partial` permet de créer une nouvelle fonction avec certains arguments pré-remplis.

Exemple :

```python
from functools import partial

def power(base, exponent):
    return base ** exponent

square = partial(power, exponent=2)
print(square(5))  # 25
```

Complexité Algorithmique (Big-O)

La **complexité algorithmique** mesure la performance d'un algorithme en fonction de la taille de l'entrée. Elle est souvent exprimée avec la notation Big-O, qui décrit la pire performance d'un algorithme.

Complexités communes :

- **O(1)** : Complexité constante. L'algorithme s'exécute en temps constant, peu importe la taille de l'entrée.
- **O(n)** : Complexité linéaire. Le temps d'exécution augmente proportionnellement à la taille de l'entrée.
- **O(n²)** : Complexité quadratique. Utilisé pour décrire des algorithmes de type boucle imbriquée.
- **O(log n)** : Complexité logarithmique. Utilisé pour décrire des algorithmes où la taille du problème est divisée à chaque étape, comme la recherche binaire.

Exemples :

O(1) : Accéder à un élément dans une liste.

```python
def get_first_element(lst):
    return lst[0]
```

O(n) : Parcourir tous les éléments d'une liste.

```python
def print_all_elements(lst):
    for elem in lst:
        print(elem)
```

O(n^2) : Boucles imbriquées.

```python
def print_all_pairs(lst):
    for i in lst:
        for j in lst:
            print(i, j)
```
