AIO25 - M1W3

GRID137

June 19, 2025

Contents

List of Figures

List of Tables

Chapter 1

Classes and Objects

1.1 What is Object-Oriented Programming?

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects." These objects encapsulate data (attributes) and methods to process the data.

1.1.1 Properies of OOP

Property	**Description**
Abstraction	Helps to hide unnecessary details and show only the essential features of an object to the user.
Inheritance	Enables code reuse by allowing a class to inherit attributes and methods from another class.
Encapsulation	Protects data from unauthorized access by restricting access to certain components.
Polymorphism	Allows objects to be treated differently based on the context, enabling flexible and dynamic behavior.

1.2 OOP in Python

1.2.1 Class

A class is a blueprint for creating objects (a particular data structure), providing initial values for state (member variables or attributes), and implementations of behavior (member functions or methods).

Coding

```
class Book:
  def __init__(self, title, author):
    self.title = title
    self.author = author
    self.rm_time = 0
    self.is_borrow = false
```

Listing 1.1: Define Book class

Explanation:

- `class Book`: Defines a class named `Book`.
- `__init__ method`: Initializes the `title`, `author`,`rm_time` and `is_borrow` attributes when a new object is created.

1.2.2 Objects

An Object is an instance of a Class. It represents a specific implementation of the class and holds its own data.

Creating object

```
class Book:
  def __init__(self, title, author):
    self.title = title
    self.author = author
    self.rm_time = 0
    self.is_borrow = false

book1 = Book("Progamming Python for beggin", "A")
book2 = Book("Data structure","B")

print(book1.title)
print(book2.author)
```

Listing 1.2: creating object

```
Progamming Python for beggin
B
```

Listing 1.3: Output object

Explanation:

- `book1 = Book("Programming Python for Beginners", "A")`: Creates an object of the `Book` class with `title` as "Programming Python for Beginners" and `author` as "A".
- `book2 = Book("Data Structure", "B")`: Creates another object of the `Book` class with `title` as "Data Structure" and `author` as "B".
- `book1.title`: Accesses the `title` attribute of the `book1` object.
- `book2.author`: Accesses the `author` attribute of the `book2` object.

1.2.3 Sefl parameter

- The self keyword is used to represent the instance of the class.
- Variables prefixed with self are the attributes of the class, while others are merely local variables of the class

```
class Book:
  def __init__(self, title, author):
    # self-prefixed attributes (instance variables)
    self.title = title
    self.author = author
    self.status = "Available"  # Default value
    # Local variable
    temp = "This is temporary"  # Only accessible within this method

  def borrow(self):
    if self.status == "Available":
    self.status = "Borrowed"
    print(f"The book ’{self.title}’ has been borrowed.")
    else:
    print(f"The book ’{self.title}’ is already borrowed.")

    # temp is not accessible here
    # print(temp)  # This will raise a NameError

book1 = Book("Data Structures", "A")
# print(book1.temp) # This will raise a AttributeError
```

1.2.4 ___init___ method

___init___ method is the constructor in Python, automatically called when a new object is created. It initializes the attributes of the class.

```
class Birthday:
  def __init__(self, day, month,year):
    self.day = day
    self.month = month
    self.year = year

birthday_peter = Birthday(1,1,2010)
print(f"{birthday_peter.day}/{birthday_peter.month}/{birthday_peter.year}")
```

Listing 1.4: ___init___ method

```
1/1/2010
```

Listing 1.5: Output ___init___ method

Explanation:

- `__init__` : Special method used for initialization.
- `self.day`, `self.month`, `self.year`: Instance attributes initialized in the constructor.

1.2.5 __call__ Method

The __call__ method in Python allows an object of a class to be called like a function. It is automatically executed when the object is followed by parentheses.

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

def __call__(self, message):
return f"{message}, {self.name}!"

greet = Greeting("Alice")
print(greet("Hello"))
```

Listing 1.6: __call__ Method

```
Hello, Alice!
```

Listing 1.7: Output __call__ method

Explanation:

- `__call__`: A special method in Python that makes an instance of a class callable like a function.
- `self.name`: An instance attribute initialized in the constructor, storing the name.
- `greet`: An instance of the `Greeting` class, initialized with the name "Alice".
- `greet("Hello")`: Invokes the __call__ method of the `greet` object, returning "Hello, Alice!".

1.3 Python Inheritance

Inheritance is a fundamental concept in object-oriented programming (OOP). It allows one class (child class) to inherit the attributes and methods of another class (parent class). This promotes code reuse and enables the creation of a hierarchical relationship between classes.

1.3.1 How Inheritance Works

When a class inherits from another:

- The child class gains access to all public and protected attributes and methods of the parent class.
- The child class can also override methods of the parent class to provide specific behavior.
- The `super()` function is used to access methods and attributes of the parent class from the child class.

1.3.2 Example: Basic Inheritance

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

  def speak(self):
    return f"{self.name} makes a sound."

# Child class
class Dog(Animal):
  def __init__(self, name, breed):
    super().__init__(name)  # Initialize the parent class
    self.breed = breed

  def speak(self):
    return f"{self.name}, the {self.breed}, barks."

# Child class
class Cat(Animal):
  def __init__(self, name, color):
    super().__init__(name)  # Initialize the parent class
    self.color = color

  def speak(self):
    return f"{self.name}, the {self.color} cat, meows."

# Create instances
dog = Dog("Buddy", "Golden Retriever")
cat = Cat("Whiskers", "white")

print(dog.speak())  # Output: Buddy, the Golden Retriever, barks.
print(cat.speak())  # Output: Whiskers, the white cat, meows.
```

Listing 1.8: Inheritance Example

1.3.3 Explanation

- `class Animal:` This is the parent class. It contains common attributes (`name`) and methods (`speak`) that all child classes can inherit.

- `super().__init__(name):` The `super()` function is used in the child class to call the parent class's constructor and initialize inherited attributes.

- `class Dog(Animal):` This is a child class inheriting from `Animal`. It adds a new attribute (`breed`) and overrides the `speak()` method.

- `class Cat(Animal):` Similar to `Dog`, this child class inherits from `Animal`, adds a new attribute (`color`), and overrides the `speak()` method.

- `dog.speak()`: Calls the overridden `speak()` method in the `Dog` class, which includes specific behavior for dogs.

- `cat.speak()`: Calls the overridden `speak()` method in the `Cat` class, which includes specific behavior for cats.

Chapter 2

OOP with Python (Custom Pytorch Class)

2.1 Delegation

2.1.1 What is Delegation?

2.1.2 Why Use Delegation?

2.2 Inheritance

2.2.1 What is Inheritance

2.2.2 Common Pitfalls when using Inheritance

2.3 When to Use Inheritance vs Delegation

Table 2.1: When to Use Inheritance vs Delegation

Scenario	Prefer Inheritance	Prefer Delegation (Composition)
Relationship	Clear "is-a" (`Dog` is-a `Animal`)	"has-a" or "uses-a" (`Car` has-a `Engine`)
Behavior Extension	You need to override or extend base behavior directly	You want to assemble behavior from multiple, independent components
Runtime Flexibility	Static relationships known at design time	Ability to swap in different implementations at runtime
Coupling Concerns	Acceptable tight coupling to parent	You need loose coupling and easier testing via mocks
Hierarchy Complexity	Simple, shallow hierarchies	Avoids deep, fragile inheritance trees
Single Responsibility	Parent and child share cohesive domain	Each class focuses on its own concern

2.4 Example Custom PyTorch Classes

Chapter 3

Database - SQL(3)

Chapter 4

OOP + Data structure (Graph and Tree)

4.1 Stack and Queue

Stack and Queue definition

Stack and Queue are special uses of List in Python, with specific constraints:

- Stack: only add/remove element from one end (LIFO).
- Queue: Add at one end, remove from the other one (FIFO).

4.1.1 Stack

Stack Visualization

Stacks return elements in the reverse order in which they are stored; that is, the most recent element to be added is returned. We call this kind of data structure last-in-first-out (LIFO).

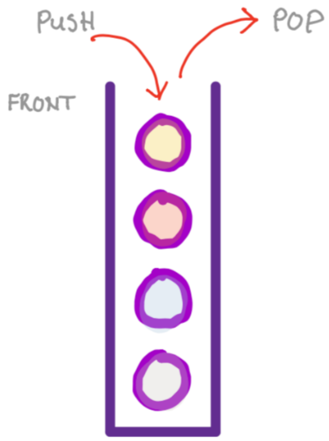

Figure 4.1: Stack visualized

Push/Pop Visualization

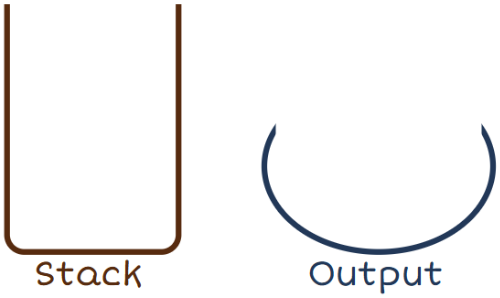

Figure 4.2: Create an Empty stack

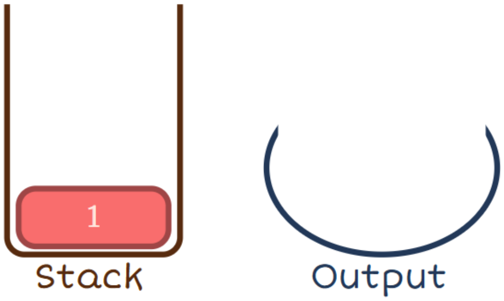

Figure 4.3: Push in the first element

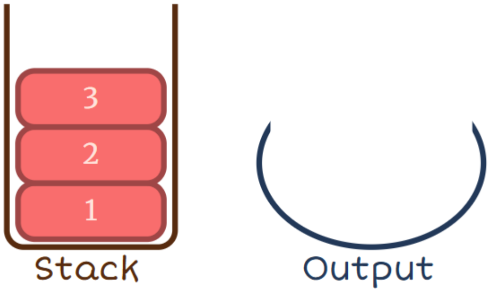

Figure 4.4: Push in the second and third element

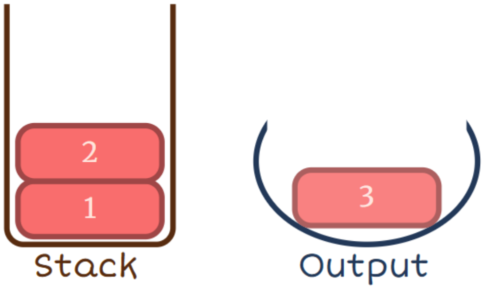

Figure 4.5: Pop out the element at the top of the stack

4.1.2 Coding

```python
class MyStack:
  def _init_(self, capacity):
    self._capacity = capacity
    self._stack = []

  def push(self, value):
    if self.is_full():
      print(’Do nothing!’)
    else:
      self._stack.append(value)

  def pop(self):
    if self.is_empty():
      print(’Do nothing’)
      return None
    else:
      return self._stack.pop()

  def print(self):
    print(self._stack)

  def is_full(self):
    return len(self._stack) == self._capacity
```

Listing 4.1: Define Stack data structure class

```python
stack1 = MyStack(5)
stack1.push(12)
stack1.push(8)
stack1.push(21)
stack1.push(33)
stack1.push(34)
stack1.push(35)
stack1.print()

//Output: Do nothing!
```

```
        [12, 8, 21, 33, 34]
```

Listing 4.2: Push and Pop example

Chapter 5

Unix and Docker