

AIO25 - M1W3

GRID137

June 19, 2025

# Contents

<b>1</b>	<b>Classes and Objects</b>	<b>2</b>
1.1	What is Object-Oriented Programming? . . . . .	2
1.1.1	Properties of OOP . . . . .	2
1.2	OOP in Python . . . . .	2
1.2.1	Class . . . . .	2
1.2.2	Objects . . . . .	3
1.2.3	Self parameter . . . . .	4
1.2.4	__init__ method . . . . .	4
1.2.5	__call__ Method . . . . .	5
1.3	Python Inheritance . . . . .	5
1.3.1	How Inheritance Works . . . . .	5
1.3.2	Example: Basic Inheritance . . . . .	6
1.3.3	Explanation . . . . .	6

## 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 Properties 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

```
1 class Book:
2     def __init__(self, title, author):
3         self.title = title
4         self.author = author
5         self.rm_time = 0
6         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

```
1 class Book:
2     def __init__(self, title, author):
3         self.title = title
4         self.author = author
5         self.rm_time = 0
6         self.is_borrow = false
7
8 book1 = Book("Programming Python for beggin", "A")
9 book2 = Book("Data structure","B")
10
11 print(book1.title)
12 print(book2.author)
```

Listing 1.2: creating object

```
1 Programming Python for beggin
2 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 Self 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

```
1 class Book:
2     def __init__(self, title, author):
3         # self-prefixed attributes (instance variables)
4         self.title = title
5         self.author = author
6         self.status = "Available" # Default value
7         # Local variable
8         temp = "This is temporary" # Only accessible within this method
9
10    def borrow(self):
11        if self.status == "Available":
12            self.status = "Borrowed"
13            print(f"The book '{self.title}' has been borrowed.")
14        else:
15            print(f"The book '{self.title}' is already borrowed.")
16
17        # temp is not accessible here
18        # print(temp) # This will raise a NameError
19
20    book1 = Book("Data Structures", "A")
21    # 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.

```
1 class Birthday:
2     def __init__(self, day, month, year):
3         self.day = day
4         self.month = month
5         self.year = year
6
7 birthday_peter = Birthday(1,1,2010)
8 print(f"{birthday_peter.day}/{birthday_peter.month}/{birthday_peter.year}")
```

Listing 1.4: `__init__` method

```
1 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.

```
1 class Greeting:
2     def __init__(self, name):
3         self.name = name
4
5     def __call__(self, message):
6         return f"{message}, {self.name}!"
7
8 greet = Greeting("Alice")
9 print(greet("Hello"))
```

Listing 1.6: `__call__` Method

```
1 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

```
1  # Parent class
2  class Animal:
3      def __init__(self, name):
4          self.name = name
5
6      def speak(self):
7          return f"{self.name} makes a sound."
8
9  # Child class
10 class Dog(Animal):
11     def __init__(self, name, breed):
12         super().__init__(name)  # Initialize the parent class
13         self.breed = breed
14
15     def speak(self):
16         return f"{self.name}, the {self.breed}, barks."
17
18 # Child class
19 class Cat(Animal):
20     def __init__(self, name, color):
21         super().__init__(name)  # Initialize the parent class
22         self.color = color
23
24     def speak(self):
25         return f"{self.name}, the {self.color} cat, meows."
26
27 # Create instances
28 dog = Dog("Buddy", "Golden Retriever")
29 cat = Cat("Whiskers", "white")
30
31 print(dog.speak())  # Output: Buddy, the Golden Retriever, barks.
32 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 OOP in Python

#### 2.1.1 Object-Oriented Programming (OOP) vs Procedural

# 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).

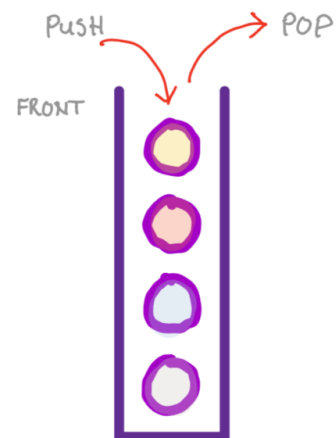


Figure 4.1: Stack visualized

## Push/Pop Visualization

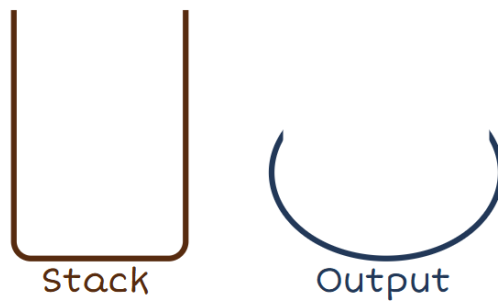


Figure 4.2: Create an Empty stack

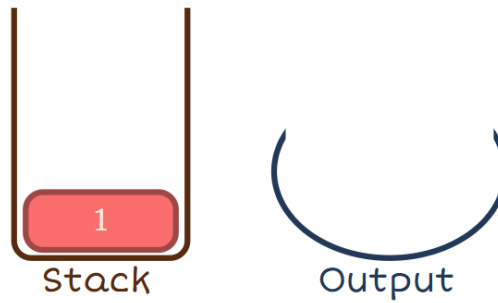


Figure 4.3: Push in the first element

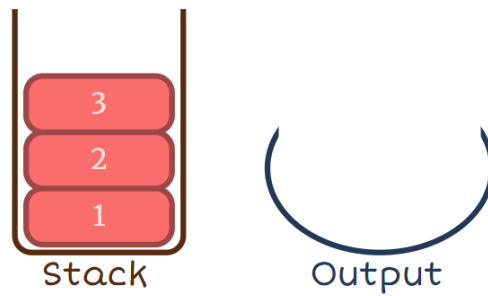


Figure 4.4: Push in the second and third element

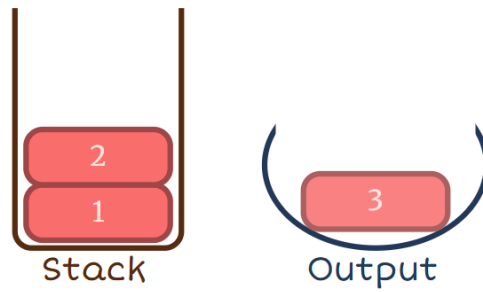


Figure 4.5: Pop out the element at the top of the stack

### 4.1.2 Coding

```
1  class MyStack:
2      def _init_(self, capacity):
3          self._capacity = capacity
4          self._stack = []
5
6      def push(self, value):
7          if self.is_full():
8              print('Do nothing!')
9          else:
10             self._stack.append(value)
11
12     def pop(self):
13         if self.is_empty():
14             print('Do nothing')
15         return None
16     else:
17         return self._stack.pop()
18
19     def print(self):
20         print(self._stack)
21
22     def is_full(self):
23         return len(self._stack) == self._capacity
```

Listing 4.1: Define Stack data structure class

```
1  stack1 = MyStack(5)
2  stack1.push(12)
3  stack1.push(8)
4  stack1.push(21)
5  stack1.push(33)
6  stack1.push(34)
7  stack1.push(35)
8  stack1.print()
9
10 //Output: Do nothing!
```

11

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

Listing 4.2: Push and Pop example

# Chapter 5

## Unix and Docker