Object Oriented Programming

August 2024

Data Science Session Housekeeping

- The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly.
- No question is daft or silly - **ask them!**
- There are **Q&A sessions** midway and at the end of the session, should you wish to ask any follow-up questions. Moderators are going to be answering questions as the session progresses as well.
- If you have any questions outside of this lecture, or that are not answered during this lecture, please do submit these for upcoming Academic Sessions. You can submit these questions here: **Questions**

Data Science Session Housekeeping cont.

- For all **non-academic questions**, please submit a query: **www.hyperiondev.com/support**
- Report a **safeguarding** incident: **www.hyperiondev.com/safeguardreporting**
- We would love your **feedback** on lectures: **Feedback on Lectures**

Learning Outcomes

- **Describe** the principles of Object-Oriented Programming (OOP) and the role of classes and objects in Python.
- **Apply** OOP concepts by creating and instantiating classes in Python, including defining attributes and methods.

Learning Outcomes

- **Design** and **implement** a basic OOP-based program to model and manipulate data, ensuring the use of best practices like naming conventions and single responsibility.

Problem Statement

In Data Science, managing complex data and designing systems that can handle growing datasets efficiently is a common challenge. We need organised, reusable, and scalable code. Storing complex data using simple data types like integers, floats and strings and data structures can be very tedious and make our code difficult to comprehend.

- How can we store more complex data efficiently?
- Is there any way that we could create our own complex data types?

HyperionDev

Lecture Overview

- Object Oriented Programming
- Classes
- Class Properties
- Methods
- Class Creation and Instantiation

HyperionDev

Object-Oriented Programming

A programming paradigm based on the concept of objects which store data in the form of attributes and code in the form of methods.

- Consider a scenario where you may want to store the information of several students in a class.
  - Each student has multiple sets of data pertaining to them.
  - There are some functions that we may need to perform for each students which involves the data pertaining to them.
- We could implement this using multiple lists or dictionaries to store all the data but this could become confusing

Object-Oriented Programming

- What if we could define a new data type: **“Student”**
- We can do this using **objects** in Python.
- **Objects** is a fundamental **building block** that represents a real-world **entity** or **concept**. It encapsulates both **data** and **behaviour**.
- In Python, everything is an object. Every entity, including data values and functions, are considered objects.
- In order to create objects, we create a “template” or “blueprint” for the object using **classes**.
- In this blueprint, we outline the different **attributes** that the object has and the different **methods** defined for the object.

Class Properties

- **Attributes** are **variables** that belong to a class. They represent the **properties or characteristics** of the class that objects can have.
- **Methods** are **functions** that belong to a class. They define the **behaviors or actions** that an object of the class can perform.

```python
class Student:
    def __init__(self, name, age, mark):
        self.name = name
        self.age = age
        self.mark = mark
```

HyperionDev

Class Properties

- We define methods in our classes the same way that we would define functions.
- To reference any of the class' attributes we use **self**.

```
def sayMyName (self):
    print("Hi, my name is " + self.name)
```

HyperionDev

Methods

- In Python, **self** has to be passed into every **instance method** as the first parameter but does not have to be included when the function is actually called. You have to reference **self** when accessing attributes.
- **Instance methods** are actions or behaviors that specific objects can perform.
- They have access to the object's data and are defined within the class.

```
def study(self):
    self.studying = True
    print("{} is studying. Please do not disturb.".format(self.name))
```

HyperionDev

Methods

- **Static methods** are standalone functions associated with a class. They are not bound to an instance of an object or class.
- With static methods, we can group together related functionalities within a class and make our code more organized and modular.

```
class MathUtils:
    @staticmethod
    def add(a, b):
        return a + b

result = MathUtils.add(5, 7)
print(result)
```

HyperionDev

Methods

- **Class methods** are functions that operate on the class itself, rather than on individual objects. They are bound to the class, not instance.
- They're useful for defining functionalities that affect the entire class, such as modifying class attributes or performing operations that involve the class as a whole.

```
@classmethod
def calculate_average (cls, num_list):
    num_sum = sum(num_list)
    n = len(num_list)
    return (num_sum/n)
```

Class Instantiation

- We use the **class** keyword to create a new class, followed by the name of the class.
- We use a **constructor function** to define anything that needs to take place when the object is first instantiated.
  - This includes any **attributes** that need to be defined, which we store using the **self** parameter, which is a reference to the object.
  - This function is called the **__init__ function** and it is called when a class is instantiated (created).

Class Instantiation

- To instantiate a class we call its constructor using the **name of the class**, followed by the **required attributes in brackets**.
- We usually store the instantiated class in a variable.
- We can access the instance methods and attributes by referencing the variable which stores the instantiated class followed by a ".".
- We can access class methods and static methods by referencing the class directly using the class' name.

Class Instantiation

```
class Student:
    def __init__(self, name, age, mark):
        self.name = name
        self.age = age
        self.mark = mark
        self.studying = False

    def study(self):
        self.studying = True
        print("{} is studying. Please do not disturb."
              .format(self.name))

student1 = Student("Zahra", 24, 89)
student1.study()
```

HyperionDev

Naming Conventions

- Python classes use the CamelCase naming convention
- Each word within the class name will start with a capital letter.
- E.g. Student, WeightExercise

```
class Student:
```

```
class WeightExercise:
```

Naming Conventions

- Give your classes meaningful and descriptive names
- Other users should already have an idea what your class is for from the name.

BAD

```
class CNum:
```

GOOD

```
class ContactNumber:
```

Single Responsibility

- Make sure your classes represent a single idea.
- If we have a person class that can have a pet we won't add all the pet attributes to the person class. We will rather create a new class.

```
class Person:

    def __init__(self, name, surname, pet_name, pet_type):
        self.name = name
        self.surname = surname
        self.pet_name = pet_name
        self.pet_type = pet_type
```

Single Responsibility

```python
class Person:

    def __init__(self, name, surname):
        self.name = name
        self.surname = surname

class Pet:

    def __init__(self, name, type):
        self.name = name
        self.type = type
```

Docstrings

- We can document our classes and class methods using docstrings in the same manner we used them with functions.
- Our class docstrings will contain a short description of the class and it's attributes.
- A method docstring will contain a short description of the methods followed by it's parameters and what will be returned.

Questions and Answers

HyperionDev

Thank you for attending

HyperionDev