Level 5 Data Engineer
Module 3 Topic 4

OOP and Data Structures

Python computer programming language

Discussion

Consider the following questions:

- What do you know about OOP?
- What data structures do you know?
- How are they useful?

Introduction

What is Object-Oriented Programming?

Object-oriented programming (OOP) is a method of structuring a program by bundling related properties and behaviours into individual objects.

Key points:

- Conceptually, objects are like the components of a system
- Objects are kind of like people, they can **do** things and they can **remember** things
- Objects can also **communicate** with each other

OOP in Python projects

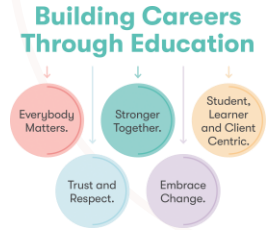
Real-world applications

OOP is used for pretty much all programming these days.

Game development

Graphical User Interface (GUI) applications

Session aim and objectives



By the end of this session, you should be able to:

- Describe the key concepts of Object-Oriented Programming (OOP) in Python
- Demonstrate knowledge of how to create classes, objects, and methods in Python
- Describe advanced OOP concepts and their use-cases

BPP

Section 1:

An Introduction to OOP

Python computer programming language

The Application of OOP

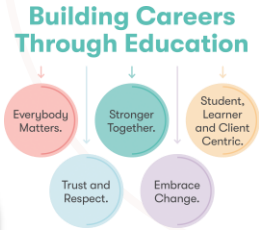
Why use OOP?

Object-Oriented Programming is useful for the following reasons:

- Simplifies code
- Models real world
- Encapsulation protects data
- Makes complex systems managable
- Modular (again)

BPP

The Benefits of OOP

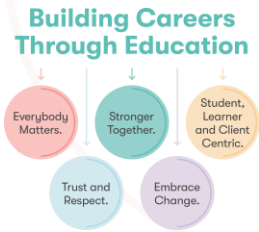


OOP is designed to organise code into reusable and **modular** components, making it easier to manage and maintain complex systems.

The benefits of the **modularity** include the following:

- Re-usability
- Organisation
- Enhanced data security

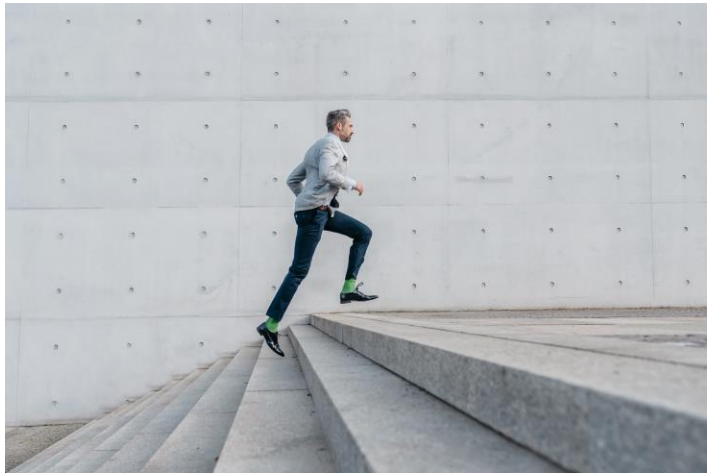
Explaining Object-Oriented Programming



What is OOP in Python?

Object-oriented Programming is an approach for modelling concrete, real-world things, like cars, as well as relations between things, like companies and employees, students and teachers, and so on.

For example, objects could include:



*A **person** with **properties** such as name, age, address, and **behaviours** such as walking, talking, breathing and running.*

*An **email** with **properties** like a recipient list, subject, and body, and **behaviours** such as adding attachments and sending.*

BPP

Defining a Class in OOP

What is a Class?

In simple terms, you can think of a **class** as a **blueprint** for creating objects.

Key points:

- Classes provide a way to create objects.
- They are a fundamental concept of OOP and allow for the creation of complex applications by modelling real-world objects or concepts

*A car **class** would define attributes such as **brand**, and **model***

Defining an Object in OOP

What is an Object?

In simple terms, an object represents a specific instance of a class.

An object in OOP would be like a real, individual car instance that you see on the road, with unique property values (attributes), for example a specific VRM (car reg.)

*A **real** car **object** would have unique values such as colour, max speed and trim*

Defining Classes and Instantiating Objects

Remember:

- Classes: In OOP, a class is a blueprint or template that defines the structure and behaviour of objects
- Objects: Objects are instances of classes, representing individual entities with their own unique data and behaviour

"Methods":

a "method" is a function that is associated with an object of a particular class. Methods are a way to define the behaviours of objects.

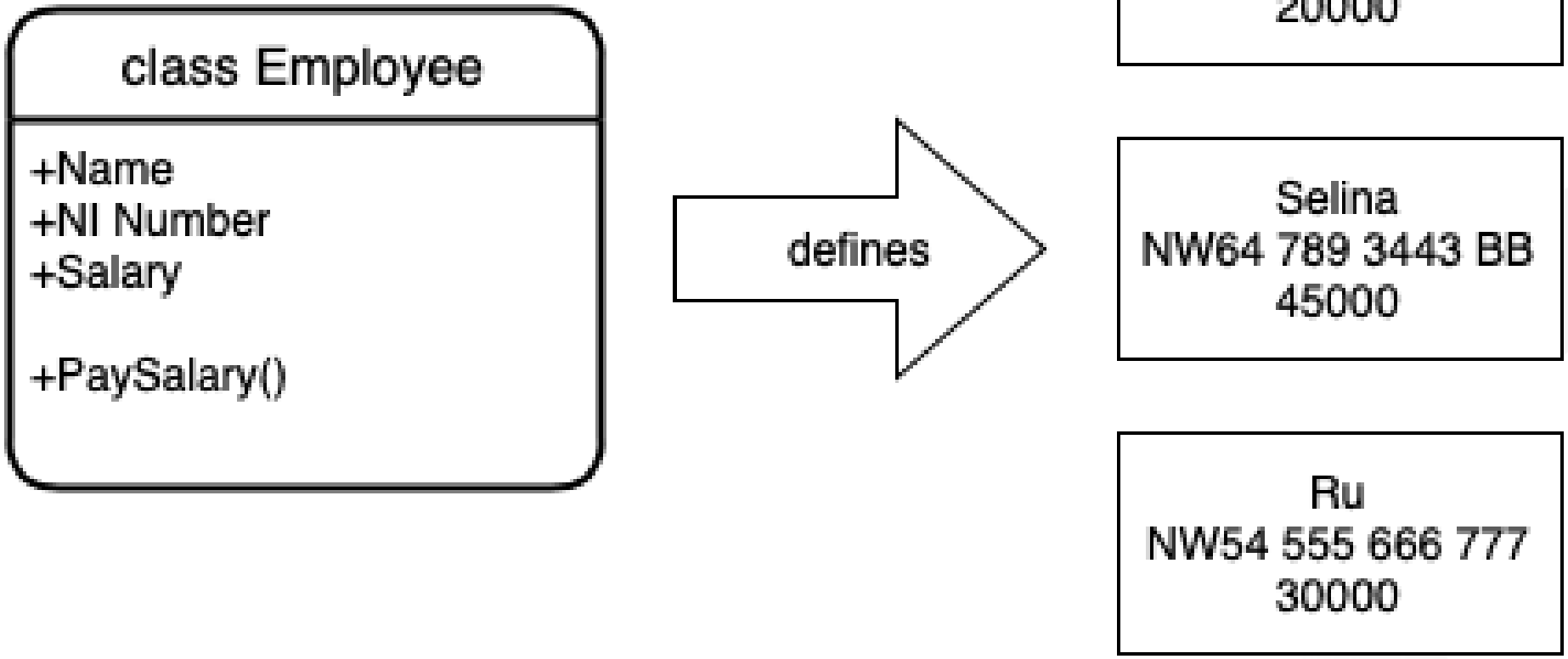
```python
class Car:
    def __init__(self, make, model, year, color):
        self.make = make
        self.model = model
        self.year = year
        self.color = color
        self.engine_status = False

    def start_engine(self):
        if not self.engine_status:
            print("Starting the engine...")
            self.engine_status = True
            print("Engine is now running.")
        else:
            print("Engine is already running.")

# Creating a car object
my_car = Car("Toyota", "Corolla", 2022, "Blue")

# Starting the engine of the car
my_car.start_engine()
```

Classes and Objects code example

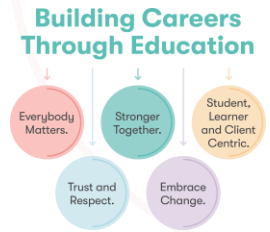
BPP

Classes can create multiple objects

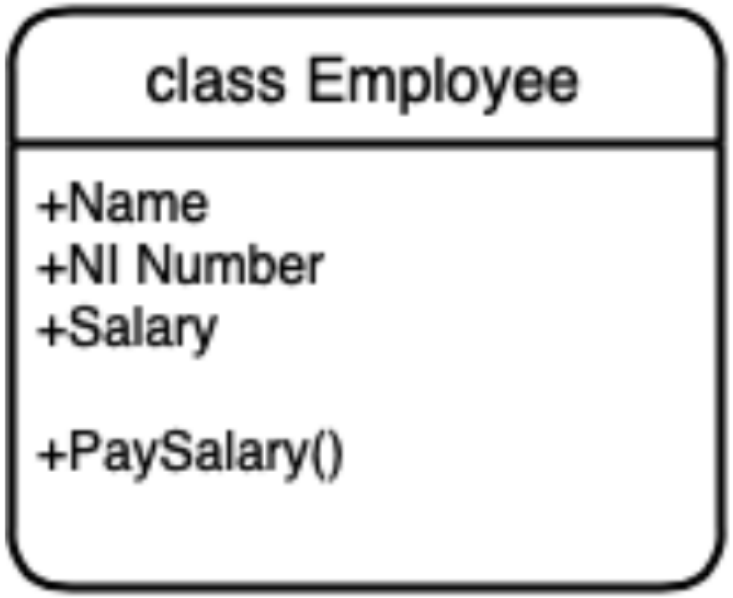
This is called object “instantiation” (or construction)



Properties and Methods

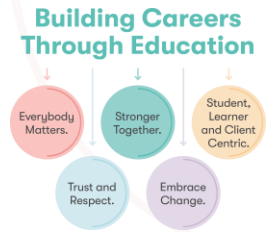


- A **property** is some data **held by** an object. Some people call properties *fields* or *attributes*.
- **Methods** are some **code** that **the object can execute**
- Methods can **manipulate or use** the field data of the object
- Properties are special **variables** that belong to each individual **object instance**
- Methods are like **functions** that are relevant to that class of object



BPP

Knowledge Check Poll



What is the role of a class in OOP?

A. A class represents a specific instance of an object with unique property values (attributes)

B. A class defines the structure and behaviour of objects, serving as a blueprint or template

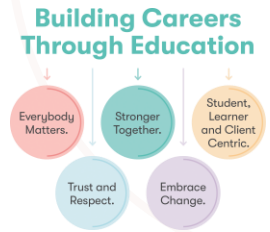
C. A class is a function that is associated with an object and defines its behaviour

D. A class allows for the creation of complex applications by organizing and encapsulating data and behaviour

Submit your responses to the chat!

BPP

Knowledge Check Poll



What is the role of a class in OOP?

A. A class represents a specific instance of an object with unique property values (attributes)

B. A class defines the structure and behaviour of objects, serving as a blueprint or template

C. A class is a function that is associated with an object and defines its behaviour

D. A class allows for the creation of complex applications by organizing and encapsulating data and behaviour

Submit your responses to the chat!

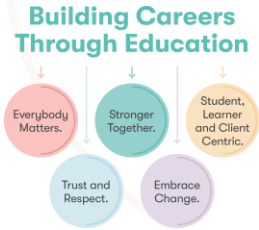
Feedback

The correct statement is **B** – A class defines the structure and behavior of objects, serving as a blueprint or template.

BPP

Object-Oriented Programming

A brief summary



Objects

- An object in code represents a real-life **thing***
- Each object contains both **data and code**

Methods

- **Methods** are code used to **manipulate object data**
- Methods and data are **encapsulated** in objects

Classes

- Objects belong to classes
- Classes **define** objects
- An object's data and methods are **defined by its class**

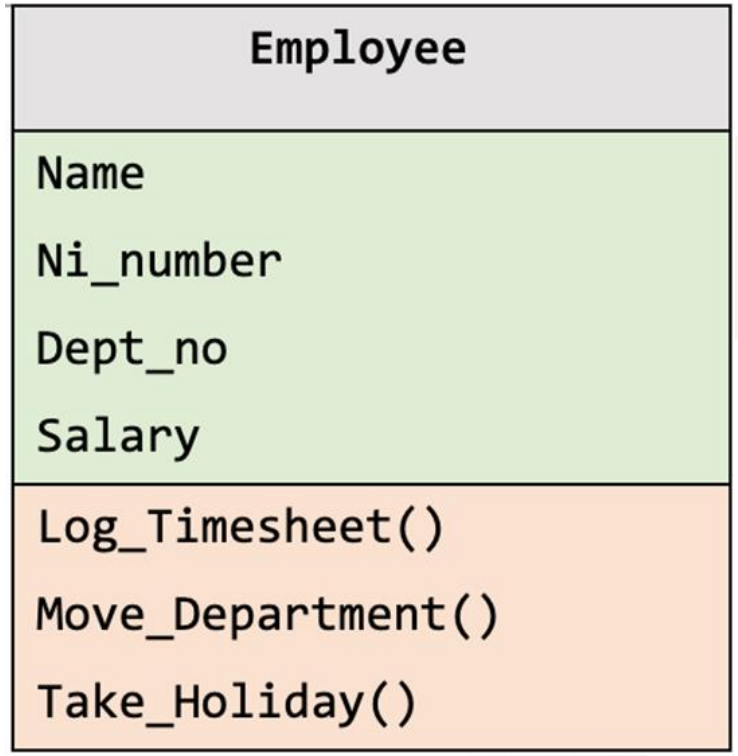
BPP

A simple diagram a Class

Deciding on properties and methods

Before we can create objects, we need to decide what properties and methods they will have and add these to a **class.**

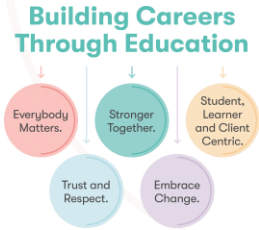
In a simple diagram, you would provide the name of the class,then its properties, and then its methods, like below:



BPP

Coding a Class

Using the “class” keyword



Once we have decided on a list of properties and methods, we can create our class using the "class" keyword in Python.

```
# Employee class encapsulates properties and methods
# of "employee" objects: could be used for an HR Application
class Employee:
    # __init__ constructor - a "dunder method"
    # which "initialises" an object when a new instance
    # of a class is created, by setting its properties
    def __init__(self, name, ni_number):
        self.name = name
        self.ni_number = ni_number
        self.salary = 0
        self.department = 0
```

BPP

Instantiating an Object of a Class

Importing class code

Having written our Employee class code, we can import it and start to create actual Employee objects.

```
>>> from HR import Employee
>>> empl1=Employee("Joe Haigh","NW54 123 4567 89")
>>> empl1.name
'Joe Haigh'
>>> empl1.ni_number
'NW54 123 4567 89'
```

BPP

Instantiating More Objects

There is no limit!

There is no practical limit to how may objects you can instantiate from a single class. Each instance now has its **own set of data:**

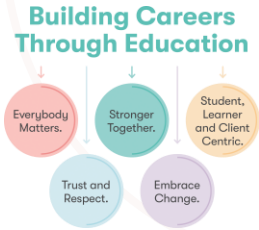
```
>>> jane = Employee("Jane Smith","NW66 233 3451 23")
>>> freddy = Employee("Fred Dredd", "NW53 234 5678 90")
>>> arno = Employee("Arno Artaud", "NW74 321 7564 32")
```

```
>>> jane.ni_number
'NW66 233 3451 23'
>>> freddy.ni_number
'NW53 234 5678 90'
>>> arno.ni_number
'NW74 321 7564 32'
```

BPP

Class Methods

Simple and complex



Functions that belong to a class are called **methods.**

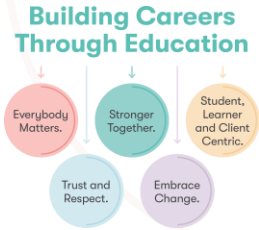
Simple methods like these, which set the values of member variables, are sometimes called "setters".

```
# set salary is a method which allows you
# to change the salary of an Employee
def set_salary(self, salary_amount):
    self.salary = salary_amount

def set_department(self, department_no):
    self.department = department_no
```

Code in file HR.py, inside of Employee class

BPP

More Complex Methods



Validation

We've changed this "setter" to prevent invalid salary amounts.

```
# set salary is a method which allows you
# to change the salary of an Employee
def set_salary(self, salary_amount):
    if salary_amount < Employee.min_salary:
        print("Salary for " + self.name + " not set: under " + str(Employee.min_salary))
    elif salary_amount > Employee.max_salary:
        print("Salary for " + self.name + " not set: over " + str(Employee.max_salary))
    else:
        self.salary = salary_amount
        print("Salary for " + self.name + " set to " + str(salary_amount))
```

BPP

Calling Methods on an Object

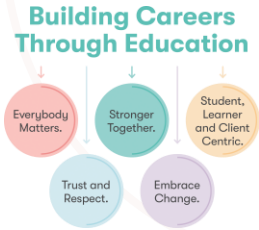
Using the set_salary() method

Now we can change an employee's salary using the set_salary() method, and guarantee that it will be within acceptable limits set via the min and max_salary properties.

```
>>> jane = Employee("Jane Smith","NW66 233 3451 23")
>>> jane.set_salary(5000)
Salary for Jane Smith not set: under 12000
>>> jane.set_salary(50000)
Salary for Jane Smith set to 50000
```

“Static” Class Members

Static members, classes and it's instances



In order for the previous code to make use of min_salary and max_salary, we need to declare these as static members of the class.

```
class Employee:
    # static members - allows us to set min and max salaries
    min_salary = 12000
    max_salary = 110000
```

BPP

Static Methods

Classes and static methods

Just as a class can have **static properties**, like min and max salaries, so it can have **static methods**.

```
@staticmethod
def get_max_salary():
    return Employee.max_salary
```

```
print("Max salary is " + str(Employee.get_max_salary()))
```

```
Max salary is 110000
```

BPP

Knowledge Check Poll

What are static members in a class?

A. Methods that belong to a class

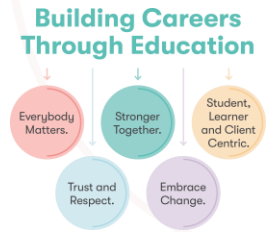
B. Methods that are used for validation

C. Properties that are shared among all instances of a class

D. Properties that can only be accessed by objects of the class

Submit your responses to the chat!

Knowledge Check Poll



What are static members in a class?

A. Methods that belong to a class

B. Methods that are used for validation

C. Properties that are shared among all instances of a class

D. Properties that can only be accessed by objects of the class

Submit your responses to the chat!

Feedback
The correct statement is **C** – Properties that are shared among all instances of a class

BPP

Section 3:

Advanced OOP Principles in Python

Python computer programming language

Section Introduction

What we will be covering?

In this section, we will be looking at advanced OOP concepts in Python programming, including:

- Dataclass decorators
- Polymorphism
- Inheritance
- Encapsulation
- Abstraction

Dataclass Decorators

What are they and why use them?

The dataclass decorator is a relatively new Python feature that automatically generates special methods, such as __init__(), __repr__(), and __eq__(), based on the class attributes.

Why use ‘Dataclass’:

- Simplifies the creation of classes with many attributes
- Automatically generates common special methods, reducing boilerplate code
- Improves code readability and maintainability

```python
from dataclasses import dataclass

@dataclass
class WeatherData:
    temperature: float
    humidity: float
    wind_speed: float

# Automatically generates __init__(), __repr__(), and __eq__() methods
weather_sample = WeatherData(25.5, 60.2, 10.3)
print(weather_sample)  # Output: WeatherData(temperature=25.5, humidity=60.2, wind_speed=10.3)
```

Dataclass example

Polymorphism

What is polymorphism and why use it?

Polymorphism allows objects of different classes to be treated as objects of a common base class.

Why use polymorphism:

- Simplifies code by providing a unified interface to multiple classes
- Enhances flexibility, allowing different objects to be used interchangeably
- Supports dynamic method dispatch, allowing the correct method to be called at runtime based on the object's actual type

```python
class Media:
    def display(self):
        pass

class Image(Media):
    def display(self):
        return "Displaying Image"

class Video(Media):
    def display(self):
        return "Playing Video"

class Audio(Media):
    def display(self):
        return "Playing Audio"

media_objects = [Image(), Video(), Audio()]

for media in media_objects:
    print(media.display())
```

Polymorphism example

Inheritance

What is inheritance and why use it?

Inheritance allows a class (subclass) to inherit attributes and methods from another class (superclass).

Why use inheritance:

- Avoids code duplication by reusing common attributes and methods
- Organises classes in a hierarchical structure
- Enables polymorphism, allowing objects of different subclasses to be treated uniformly

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

    def attack(self):
        pass

class Warrior(Character):
    def attack(self):
        return "Warrior slashes with a sword!"

class Mage(Character):
    def attack(self):
        return "Mage casts a fireball!"

player1 = Warrior("Warrior1", 100)
player2 = Mage("Mage1", 80)

print(player1.attack())  # Output: "Warrior slashes with a sword!"
print(player2.attack())  # Output: "Mage casts a fireball!"
```

Inheritance example

Encapsulation

What is encapsulation and why use it?

Encapsulation is the principle of bundling data (attributes) and methods (behaviours) that operate on the data within a class.

Why use encapsulation:

- Protects data from unauthorised access and modifications
- Allows validation and control over data access, ensuring data consistency
- Enhances code maintainability by hiding internal implementation details

```python
class BankAccount:
    def __init__(self, initial_balance):
        self._balance = initial_balance

    def deposit(self, amount):
        if amount > 0:
            self._balance += amount

    def withdraw(self, amount):
        if 0 < amount <= self._balance:
            self._balance -= amount

    def get_balance(self):
        return self._balance

account = BankAccount(1000)
account.deposit(500)
account.withdraw(200)
print(account.get_balance())  # Output: 1300
```

Knowledge Check Poll

Which advanced OOP principles allows a class to inherit attributes and methods from another class, promoting code reusability and creating a hierarchy of classes with shared functionalities?

A. Polymorphism

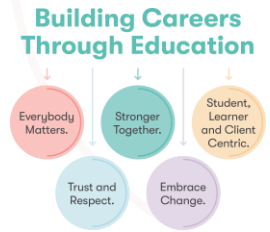
B. Encapsulation

C. Inheritance

D. Abstraction

Submit your responses to the chat!

Knowledge Check Poll



Which advanced OOP principles allows a class to inherit attributes and methods from another class, promoting code reusability and creating a hierarchy of classes with shared functionalities?

A. Polymorphism

B. Encapsulation

C. Inheritance

D. Abstraction

Submit your responses to the chat!

Feedback
The correct statement is **C** – Inheritance.

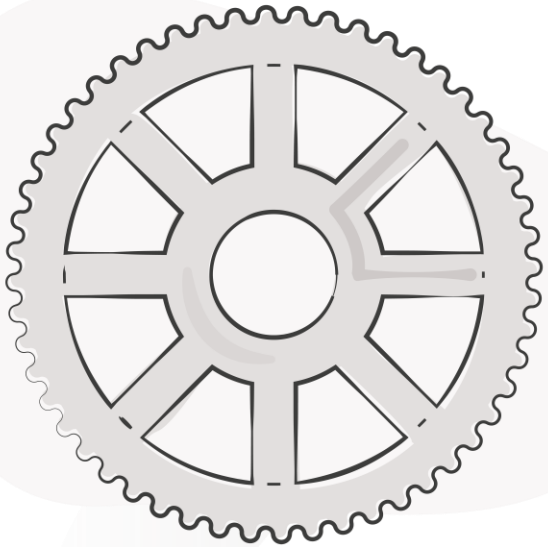
Practical application

Notebook activity

Your instructor will now walk you through the associated Python notebook for this topic.

This file can be found at the following link:

Python notebook link



Practical application

Google colab

BPP

Thank you

Do you have any questions, comments, or feedback?

How confident do you now feel about your knowledge of Object-Oriented Programming following this webinar?

- **A:** Very confident
- **B:** More confident than before this webinar
- **C:** What was this webinar session even about?!

Submit your responses to the chat!