CoGrammar

OOP: Revision

SKILLS FOR LIFE
SKILLS BOOTCAMPS

Department for Education

Software Engineering Lecture Housekeeping

- The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly. **(FBV: Mutual Respect.)**
- 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 Open Classes. You can submit these questions here: **Open Class Questions**

Software Engineering Lecture 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**

Lecture Objectives

Object Oriented Programming

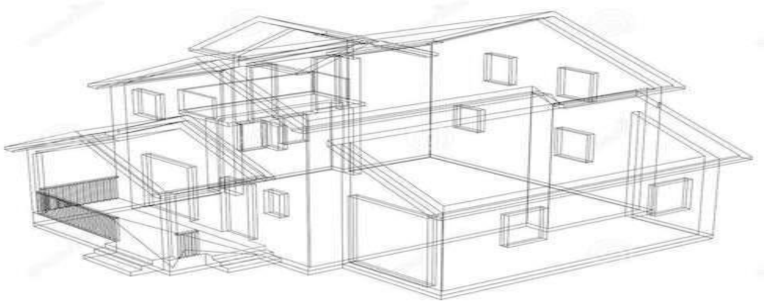
1. Class Components
2. Encapsulation
3. Abstraction
4. Inheritance
5. Polymorphism

What is Object Oriented Programming ?

OOP is a way of organising and structuring code around objects, which are self-contained modules that contain both data and instructions that operate on that data.

Classes and Objects

A class is a blueprint or template for creating objects. It defines the attributes and methods that all objects of that class will have.



An object is an instance of a class. Objects are created based on the structure defined by the class.

Attributes

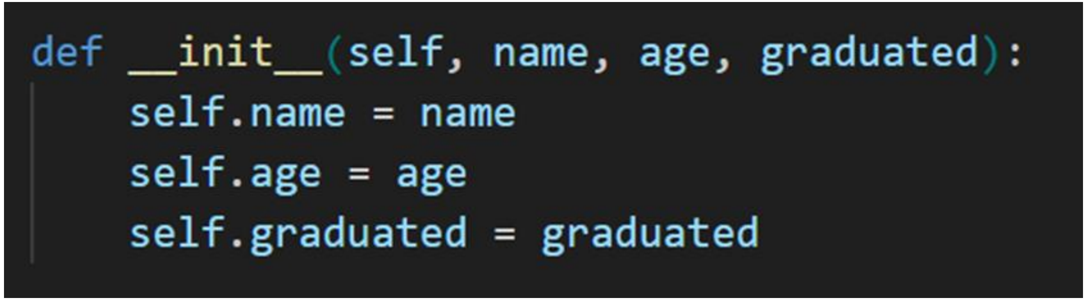
- Attributes are values that define the characteristics associated with an object.
- They define the state of an object and provide information about its current condition.
- For a class named ‘House’, some relevant attributes could be:
  - Number of bedrooms
  - Year built

Methods (Behaviours)

- Methods, also known as behaviours, define the actions or behaviours that objects can perform.
- They encapsulate the functionality of objects and allow them to interact with each other and the outside world.
- For a class named ‘House’, some relevant method could be:
  - set_location(): Allows updating the location of the house

Constructor

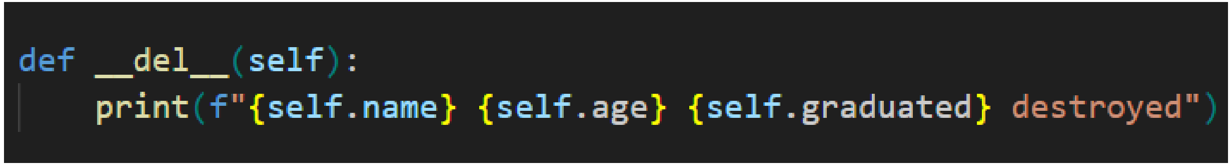
- A constructor is a special method that gets called when an object is instantiated. It is used to initialize the object's attributes.



```
def __init__(self, name, age, graduated):
    self.name = name
    self.age = age
    self.graduated = graduated
```

Destructor

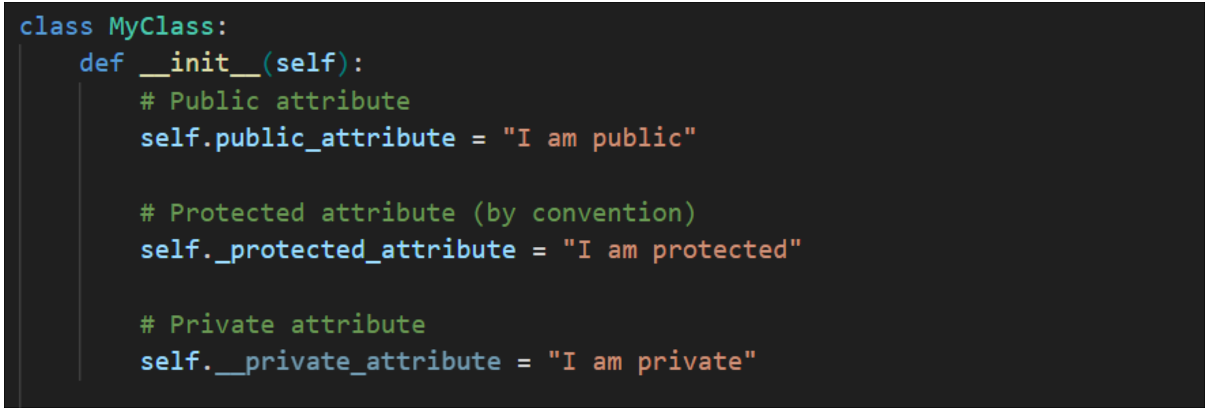
- A destructor is a special method that gets called when an object is about to be destroyed. It is used to perform cleanup operations.



```python
def __del__(self):
    print(f"{self.name} {self.age} {self.graduated} destroyed")
```

Access Control - Attributes

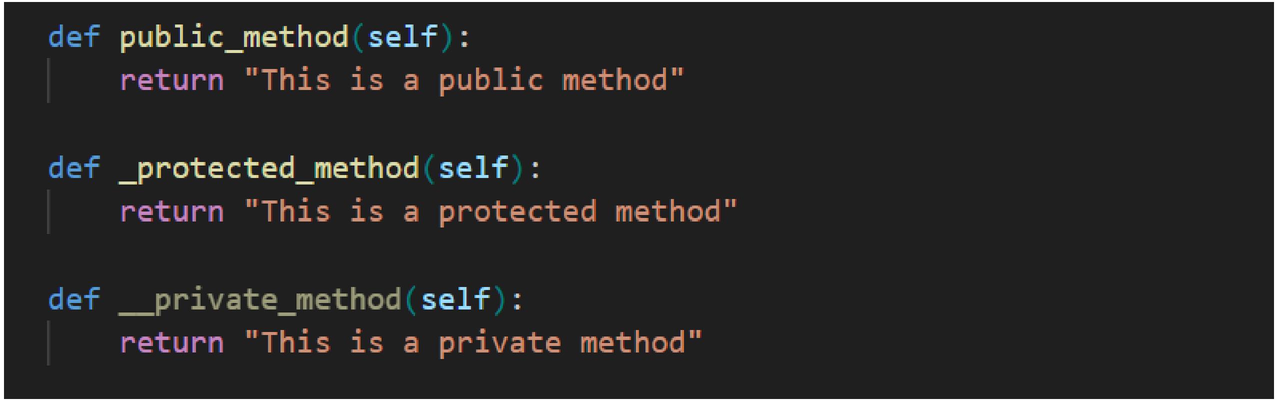
- Access control mechanisms (public, protected, private) restrict or allow the access of certain attributes with in a class.



```
class MyClass:
    def __init__(self):
        # Public attribute
        self.public_attribute = "I am public"

        # Protected attribute (by convention)
        self._protected_attribute = "I am protected"

        # Private attribute
        self.__private_attribute = "I am private"
```

Access Control - Methods

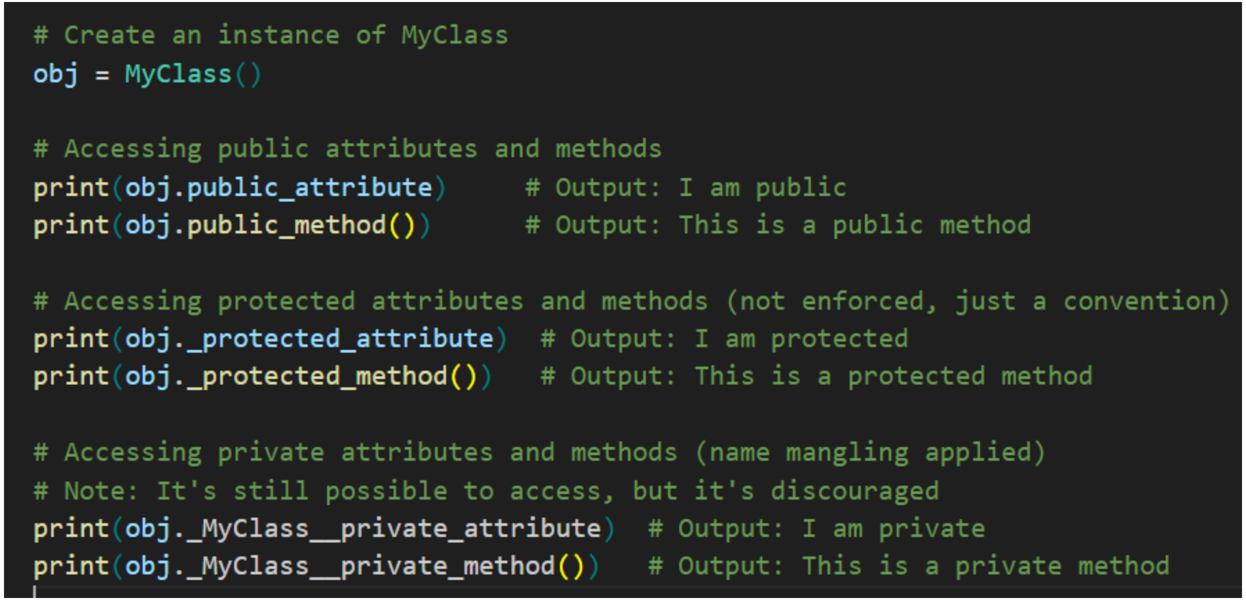
- Access control mechanisms (public, protected, private) can also restrict or allow the access of certain methods with in a class.



```python
def public_method(self):
    return "This is a public method"

def _protected_method(self):
    return "This is a protected method"

def __private_method(self):
    return "This is a private method"
```

Access Control

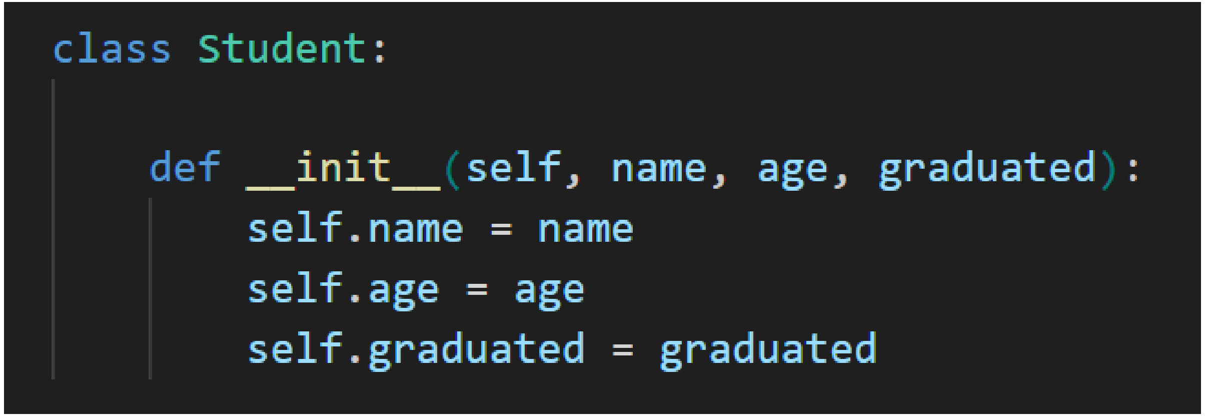
(Accessing the Attributes & Methods)



```python
# Create an instance of MyClass
obj = MyClass()

# Accessing public attributes and methods
print(obj.public_attribute)     # Output: I am public
print(obj.public_method())      # Output: This is a public method

# Accessing protected attributes and methods (not enforced, just a convention)
print(obj._protected_attribute)  # Output: I am protected
print(obj._protected_method())   # Output: This is a protected method

# Accessing private attributes and methods (name mangling applied)
# Note: It's still possible to access, but it's discouraged
print(obj._MyClass__private_attribute)  # Output: I am private
print(obj._MyClass__private_method())   # Output: This is a private method
```

Creating a Class

- __init__ ( ) method is called when the class is instantiated.

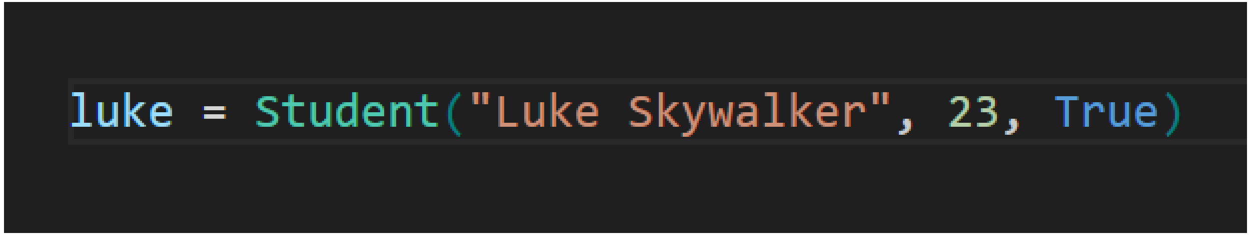


```python
class Student:

    def __init__(self, name, age, graduated):
        self.name = name
        self.age = age
        self.graduated = graduated
```

https://www.freecodecamp.org/news/python-write-to-file-open-read-append-and-other-file-handling-functions-explained/

Class Instantiation

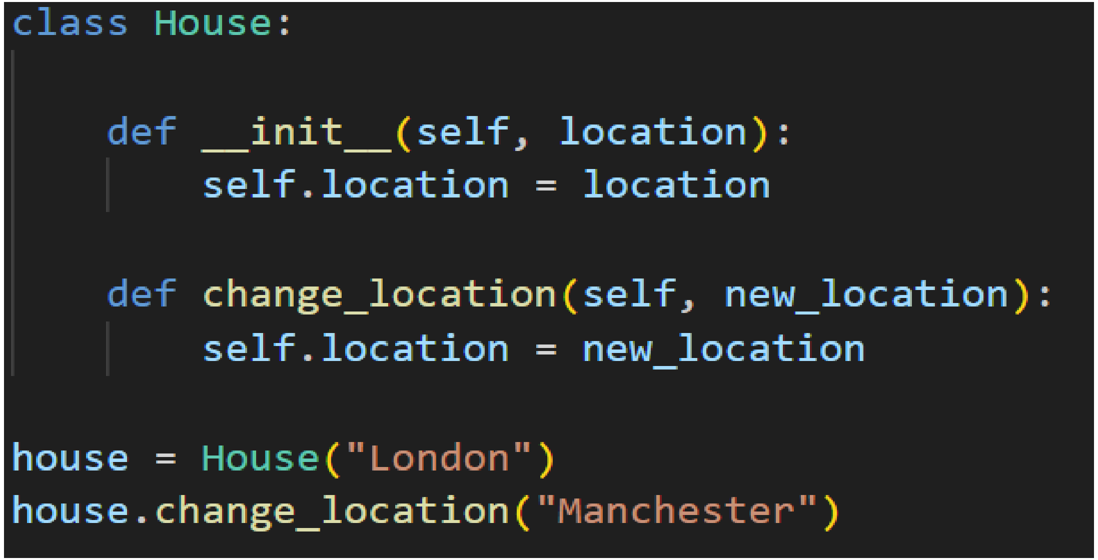
- This Class takes in three values: a name, age and graduation status.
- When you instantiate a class, you create an instance or an object of that class.



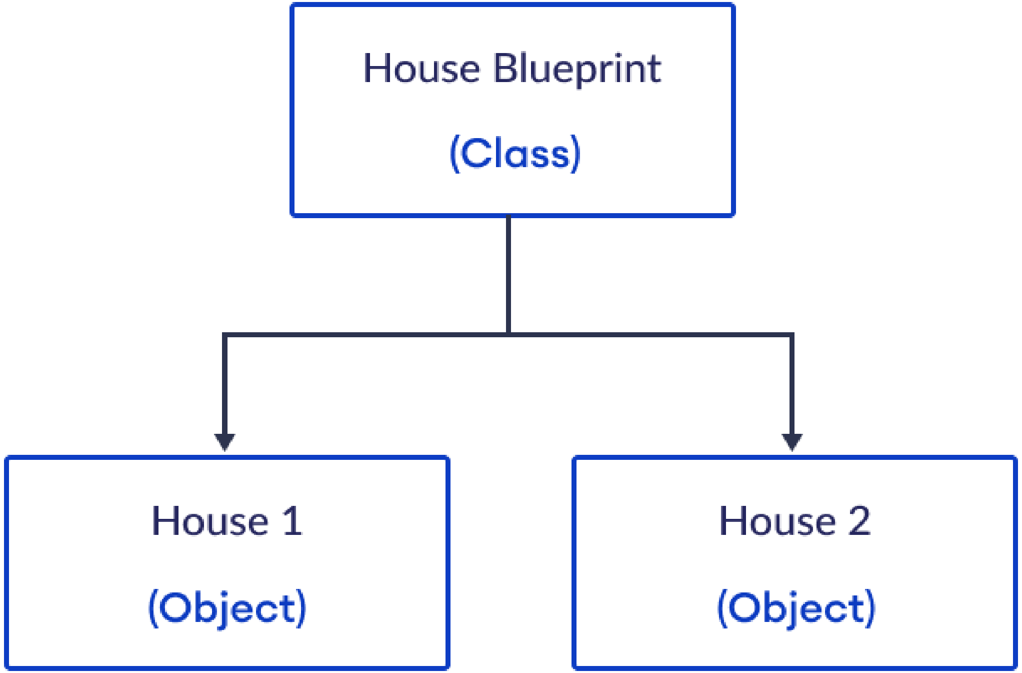
```python
luke = Student("Luke Skywalker", 23, True)
```

Creating and Calling Methods

- Change_location( ) method is called below:



```
class House:

    def __init__(self, location):
        self.location = location

    def change_location(self, new_location):
        self.location = new_location

house = House("London")
house.change_location("Manchester")
```



House Blueprint

(Class)

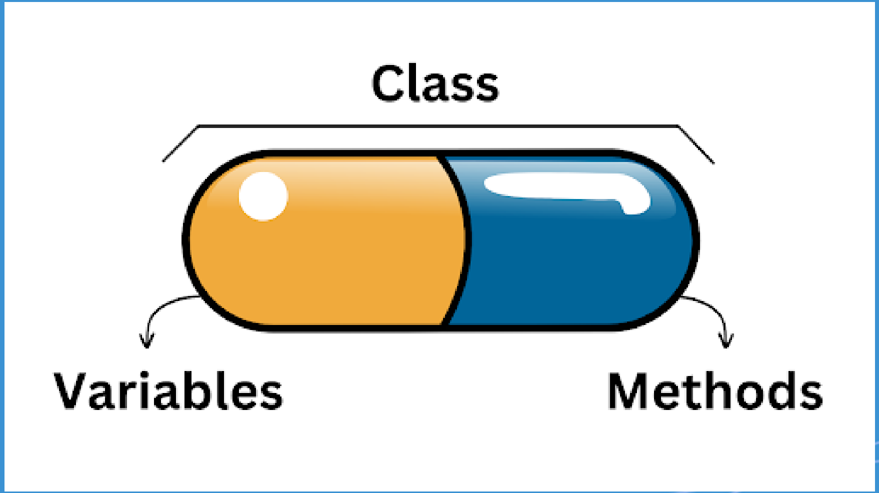
House 1

(Object)

House 2

(Object)

Encapsulation



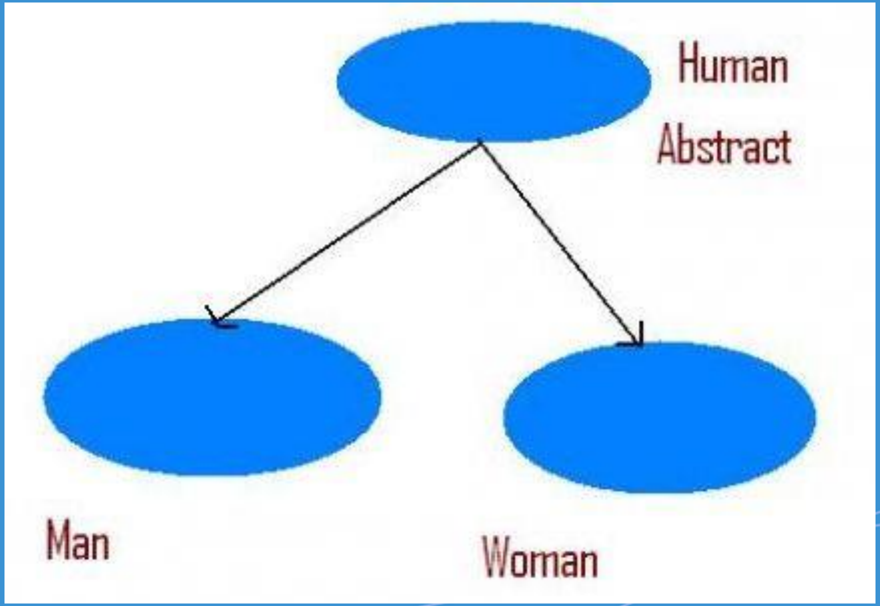
What is Encapsulation?

- Encapsulation can be likened to a protective shell that guards an object’s internal state against unintended interference and misuse. By wrapping data (attributes) and behaviours (methods) within classes and restricting access to them, encapsulation ensures a controlled interface for interaction with an object.

Why Encapsulation?

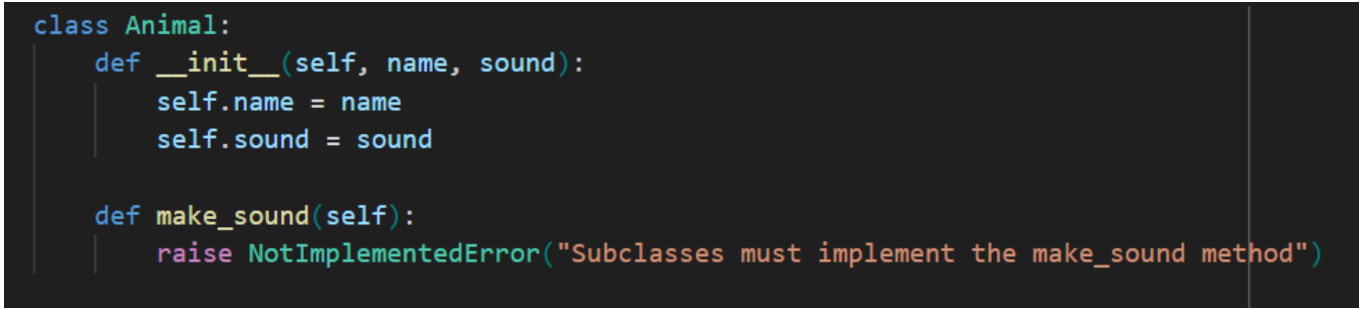
- The primary goal of encapsulation is to reduce complexity and increase reusability. By hiding the internal workings of objects, developers can simplify interactions, making them more intuitive. This abstraction layer also enhances modularity, allowing for more flexible and scalable codebases.

Abstraction



What is Abstraction?

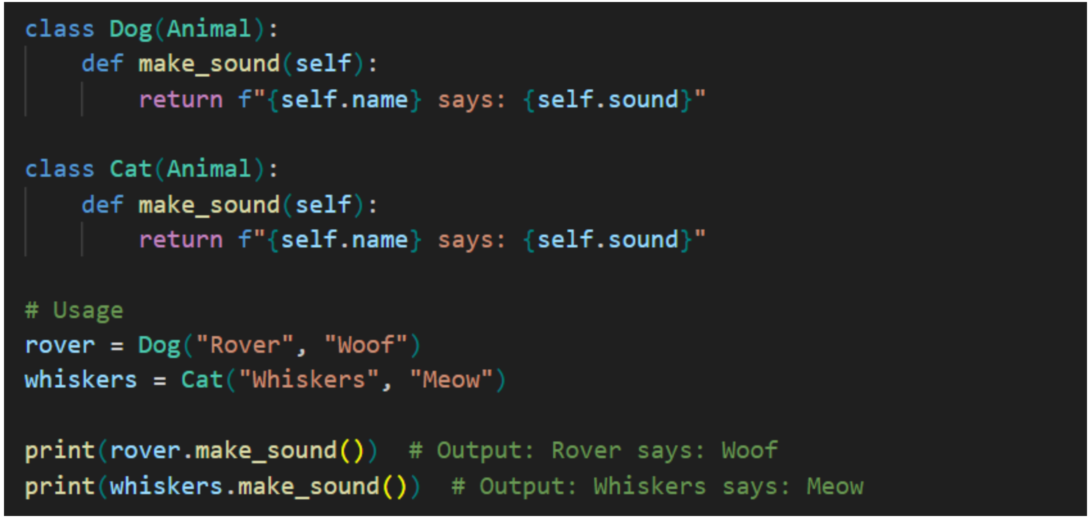
- Abstract classes cannot be instantiated, and they often define abstract methods that must be implemented by concrete subclasses.



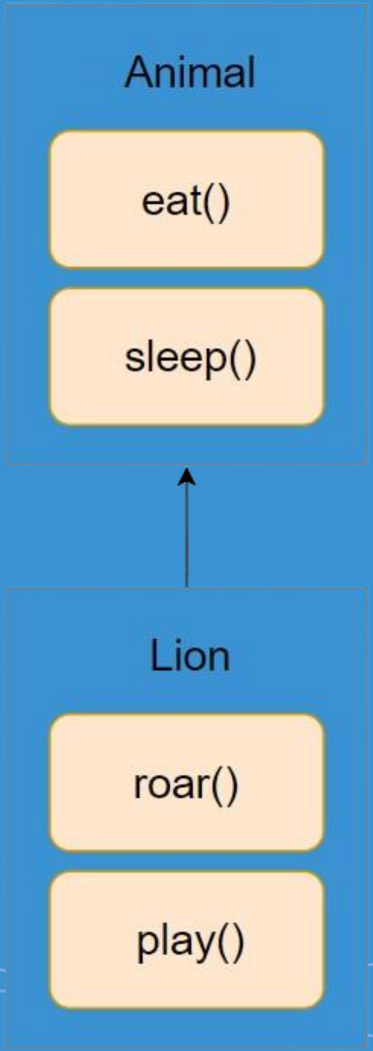
```python
class Animal:
    def __init__(self, name, sound):
        self.name = name
        self.sound = sound

    def make_sound(self):
        raise NotImplementedError("Subclasses must implement the make_sound method")
```

What is Abstraction?

- Concrete classes provide concrete (implemented) versions of the abstract method (make_sound) defined in the abstract class.



```
class Dog(Animal):
    def make_sound(self):
        return f"{self.name} says: {self.sound}"

class Cat(Animal):
    def make_sound(self):
        return f"{self.name} says: {self.sound}"

# Usage
rover = Dog("Rover", "Woof")
whiskers = Cat("Whiskers", "Meow")

print(rover.make_sound())  # Output: Rover says: Woof
print(whiskers.make_sound())  # Output: Whiskers says: Meow
```



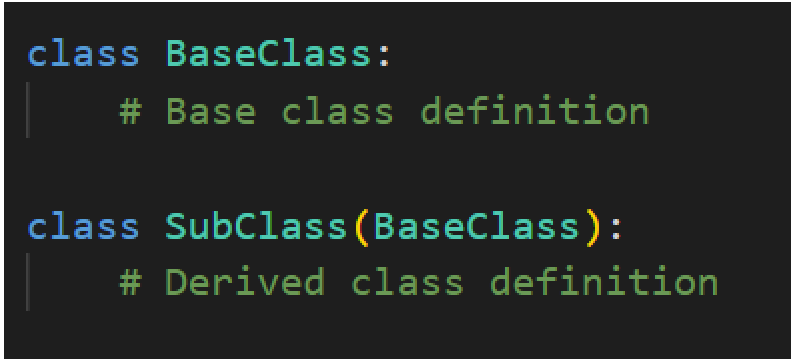
Inheritance

What is Inheritance?

- Sometimes we require a class with the same attributes and properties as another class but we want to extend some of the behaviour or add more attributes.
- Using inheritance we can create a new class with all the properties and attributes of a base class instead of having to redefine them.

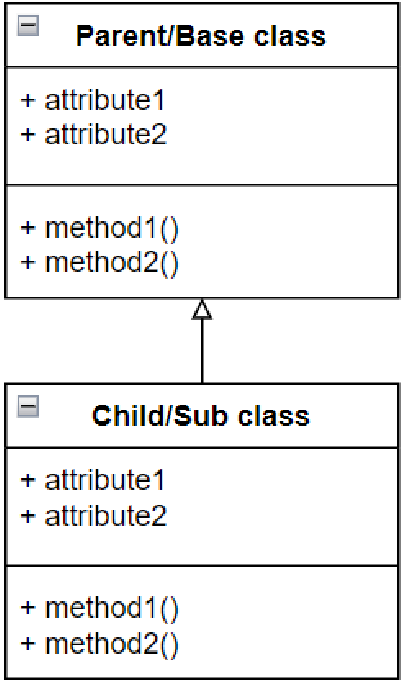
Inheritance

- Parent/Base class
  - The parent or base class contains all the attributes and properties we want to inherit.
- Child/Subclass
  - The sub class will inherit all of its attributes and properties from the parent class.



```python
class BaseClass:
    # Base class definition

class SubClass(BaseClass):
    # Derived class definition
```

Inheritance Illustrated!



Parent/Base class

+ attribute1
+ attribute2

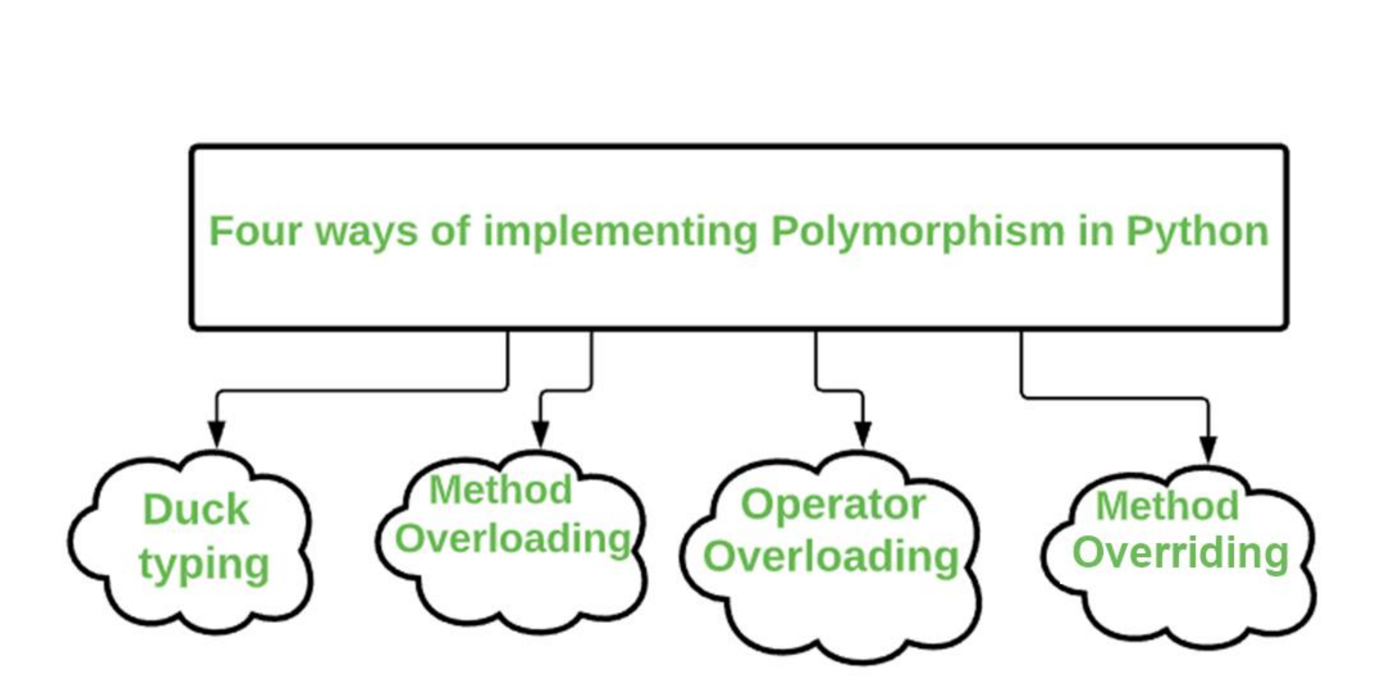
+ method1()
+ method2()

Child/Sub class

+ attribute1
+ attribute2

+ method1()
+ method2()

Polymorphism



Four ways of implementing Polymorphism in Python

Duck typing

Method Overloading

Operator Overloading

Method Overriding

Method Overriding

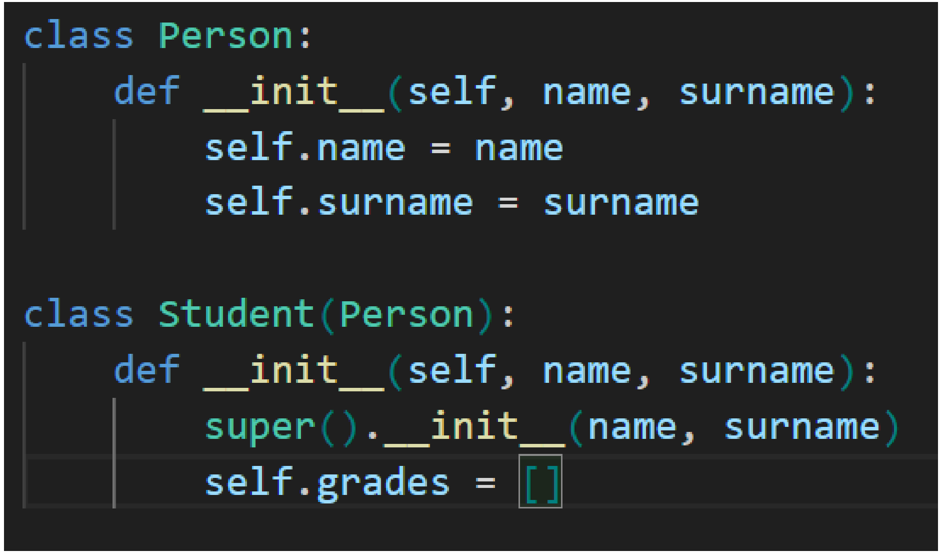
- We can override methods in our subclass to either extend or change the behaviour of a method.
- To apply method overriding you simply need to define a method with the same name as the method you would like to override.
- To extend functionality of a method instead of completely overriding we can use the super() function.

Super()

- The super() function allows us to access the attributes and properties of our Parent/Base class.
- Using super() followed by a dot “.” we can call to the methods that reside inside our base class.
- When extending functionality of a method we would first want to call the base class method and then add the extended behaviour.

Method Overriding and Super()

Here we call __init__() from the Person class to set the values for the attributes “name” and “surname”.



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

class Student(Person):
    def __init__(self, name, surname):
        super().__init__(name, surname)
        self.grades = []
```

Operator Overloading

- **Commonly Used Special Methods for Operator Overloading:**

__add__(self, other): Implement behaviour for the + operator.

__sub__(self, other): Implement behaviour for the - operator.

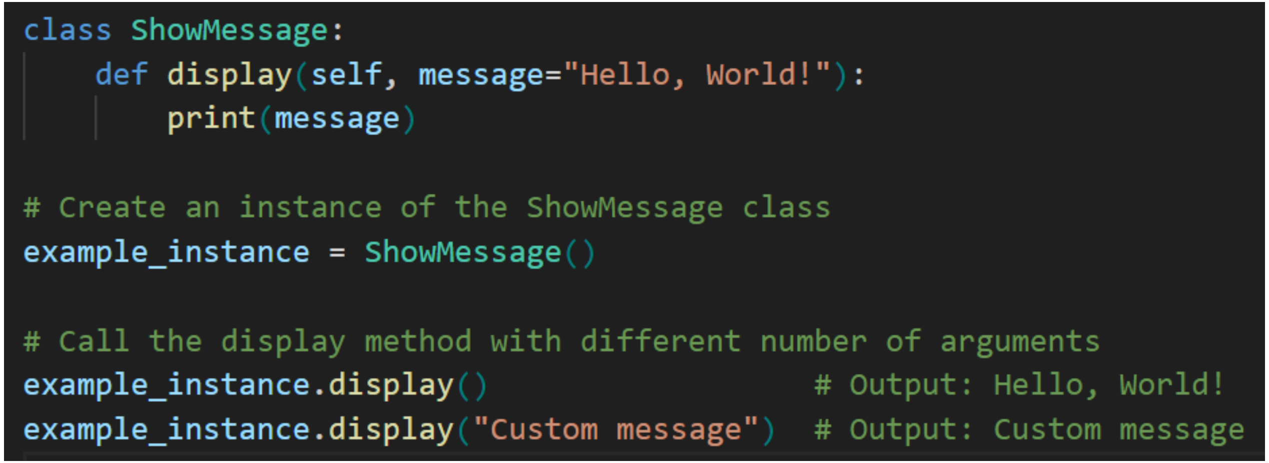
__mul__(self, other): Implement behaviour for the * operator.

__truediv__(self, other): Implement behaviour for the / operator.

__eq__(self, other): Implement behaviour for the equality (==) operator.

Method Overloading

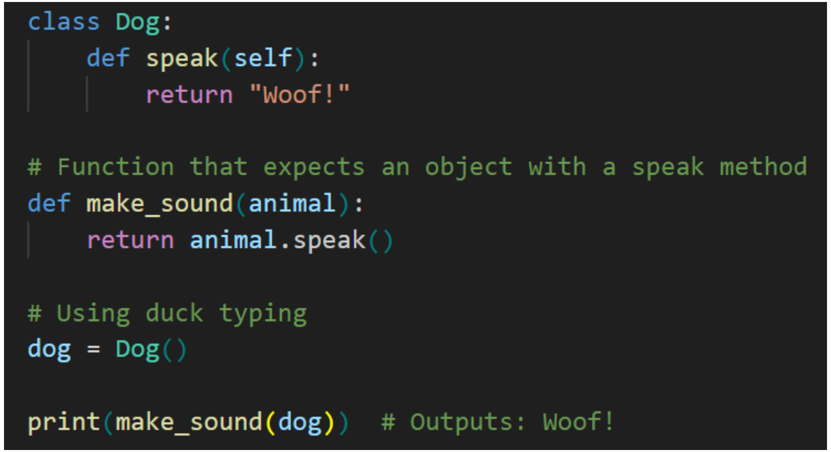
- In Python, method overloading is not supported in the same way as in some other programming languages like Java or C++. However, you can achieve similar behaviour using default values for function parameters.



```
class ShowMessage:
    def display(self, message="Hello, World!"):
        print(message)

# Create an instance of the ShowMessage class
example_instance = ShowMessage()

# Call the display method with different number of arguments
example_instance.display()                  # Output: Hello, World!
example_instance.display("Custom message")  # Output: Custom message
```

Duck Typing

- Duck typing is where the type or class of an object is less important than the methods or properties it possesses.
- The term "duck typing" comes from the saying, "If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.“



```python
class Dog:
    def speak(self):
        return "Woof!"

# Function that expects an object with a speak method
def make_sound(animal):
    return animal.speak()

# Using duck typing
dog = Dog()

print(make_sound(dog))  # Outputs: Woof!
```

CoGrammar

Questions around classes

CoGrammar

Thank you for joining

CoGrammar