Welcome to this CoGrammar session:

OOP - Classes Revision

The session will start shortly...

Questions? Drop them in the chat.
We'll have dedicated moderators
answering questions.

Software Engineering Session Housekeeping

- The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly. **(Fundamental British Values: Mutual Respect and Tolerance)**
- 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

Software Engineering 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

Software Engineering Session Housekeeping cont.

- "Please check your spam folders for any important communication from us. If you have accidentally unsubscribed, please reach out to your support team.“
- Rationale here: Career Services, Support, etc will send emails that contain NB information as we gear up towards the end of the programme. Students may miss job interview opportunities, etc.

Skills Bootcamp
8-Week Progression Overview

✓ Criterion 3: Course Progress

- ***Completion:*** All mandatory tasks, including Build Your Brand and resubmissions by study period end
- *Interview Invitation:* Within 4 weeks post-course
- *Guided Learning Hours:* Minimum of 112 hours by support end date (10.5 hours average, each week)

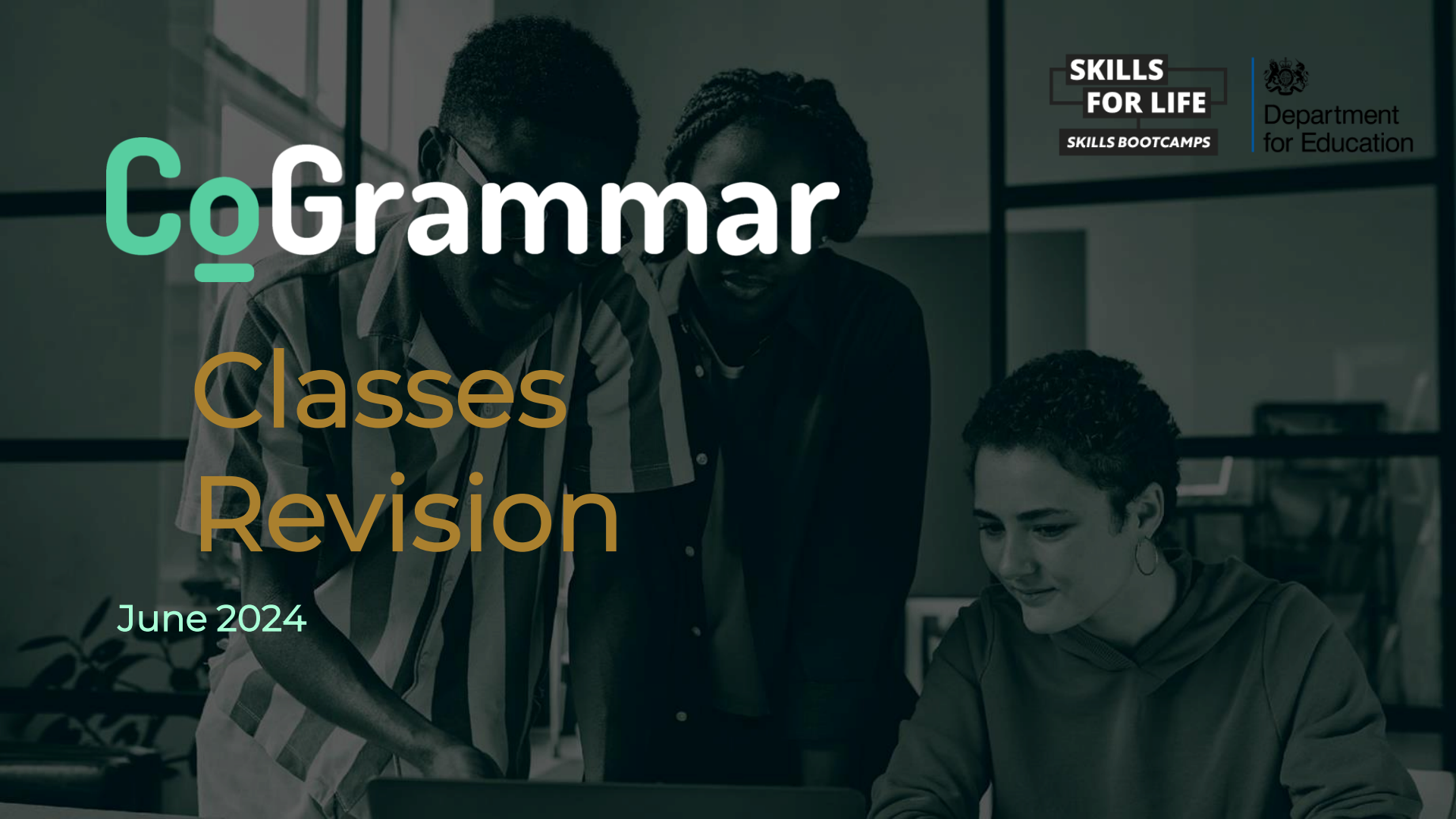
✓ Criterion 4: Demonstrating Employability

- *Final Job or Apprenticeship Outcome:* Document within 12 weeks post-graduation
- *Relevance:* Progression to employment or related opportunity

CoGrammar

Learning Outcomes

- Identify the Components of a Class
- Explain the Principle of Encapsulation
- Apply the Concept of Abstraction
- Demonstrate the Use of Inheritance
- Implement Polymorphism in Class Design



SKILLS FOR LIFE
SKILLS BOOTCAMPS

Department for Education

CoGrammar

Classes Revision

June 2024

What is Object Oriented Programming?

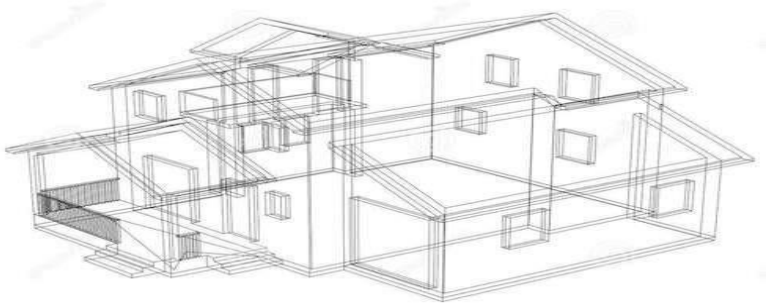
- OOP is a way of organising and structuring code around objects, which are self-contained modules that contain both data and instructions that operate on that data.

Classes

CoGrammar

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 initialise 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 clean-up operations.

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

Creating a Class

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

```
class Student:

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

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.

```
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")
```

Access Control - Attributes

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

```python
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.

```
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"
```

Applying the Access Control

```
# 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
```

Encapsulation

CoGrammar

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

CoGrammar

What is Abstraction?

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

```
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")
```

Implementing Abstraction

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

```python
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

CoGrammar

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.

Implementing Inheritance

- Parent/Base class
  - The parent or base class contains all the attributes and properties we want to inherit.

```
class BaseClass:
    # Base class definition

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

- Child/Subclass
  - The sub class will inherit all of its attributes and properties from the parent class.

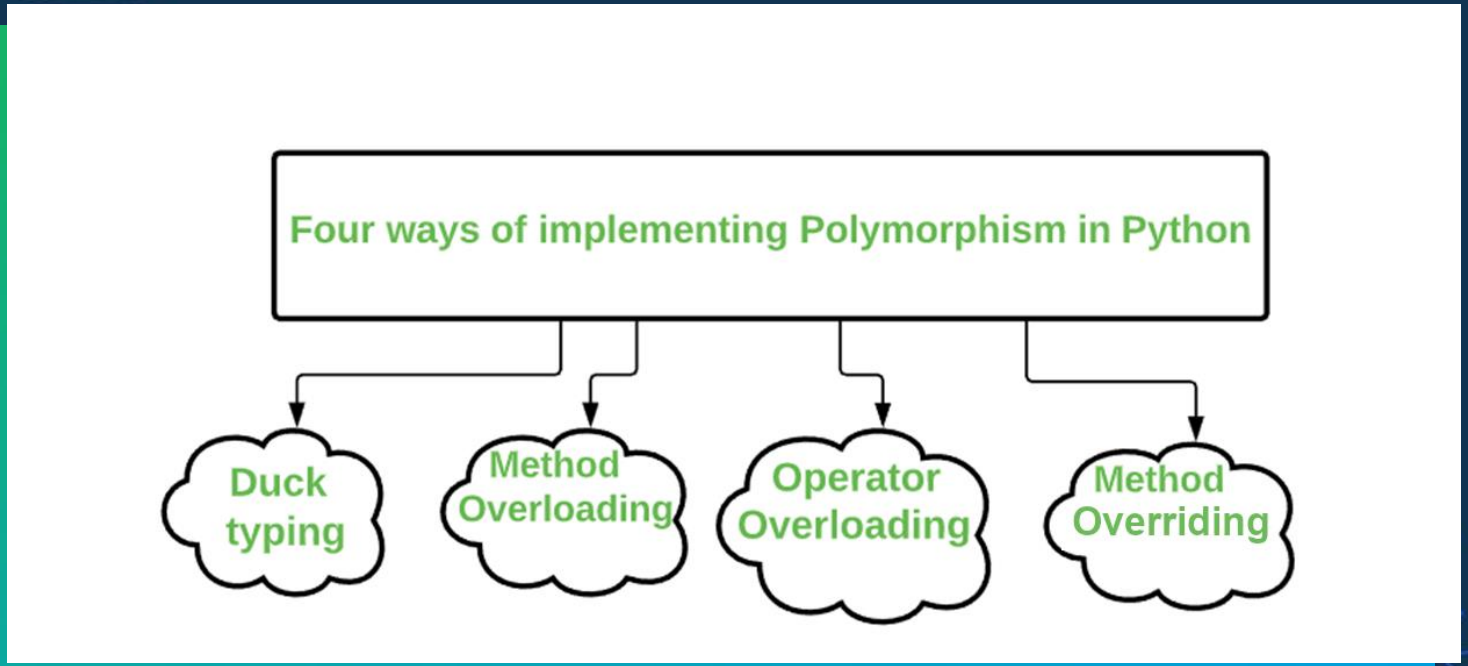
Polymorphism

CoGrammar

What is Polymorphism?

- Polymorphism refers to the ability of different objects to respond to the same message or method call in different ways.
- This allows objects of different classes to be treated as objects of a common superclass.

Implementing Polymorphism



Poly: 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() Function

- 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 copy __init__() from the parent class to set the values for the attributes “name” and “surname” and then extend it with attribute “grade”.

```
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 = []
```

Dunder Methods

- Double underscore methods are special methods in Python also known as magic methods.
- They enable customisation of object behaviour for built-in operations, providing a way of overriding built-in methods.
- Dunder methods like __str__, __len__, __iter__, and __getitem__ enable objects to exhibit polymorphic behaviour by defining their behaviour for visual presentation, length calculation, iteration, and indexing respectively.

Implementing Dunder Methods

- Commonly Used Dunder Methods for Method Overriding:
  - __init__: Constructor, initialises objects
  - __str__: Returns string representation for end-users
  - __len__: Returns the length of the object
  - __getitem__: Enables indexing and slicing
  - __iter__, __next__: Enables iteration over objects
  - __contains__: Enables membership testing using ‘in’
  - __call__: Enables objects to be callable like functions

Dunder Methods Example

```python
class CustomList:
    def __init__(self, items):
        self.items = items

    def __str__(self):
        return str(self.items)  # Customise string representation

    def __len__(self):
        return len(self.items)  # Customise behaviour for len() function

    def __getitem__(self, index):
        return self.items[index]  # Enable indexing and slicing

    def __contains__(self, item):
        return item in self.items  # Enable membership testing using 'in'

# Usage
cl = CustomList([1, 2, 3, 4, 5])
print(cl)          # Output: [1, 2, 3, 4, 5] (due to __str__)
print(len(cl))     # Output: 5 (due to __len__)
print(cl[0])       # Output: 1 (due to __getitem__)
print(3 in cl)     # Output: True (due to __contains__)
```

Poly: Operator Overloading

- Operator overloading allows custom behaviour for standard operators (like +, -, *, etc.) when they are used with user-defined objects.
- This is achieved by defining special methods in the class, such as __add__ for the + operator.
- Methods that are used for operator overloading are also part of dunder methods.

Operators for Overloading

- Commonly Used Special Methods for Operator Overloading:
  - __add__(self, other): Behaviour for the (+) operator.
  - __sub__(self, other): Behaviour for the (-) operator.
  - __mul__(self, other): Behaviour for the (*) operator.
  - __truediv__(self, other): Behaviour for the (/) operator.
  - __eq__(self, other): Behaviour for the (==) operator.

Poly: Method Overloading

- The creation of multiple methods with the same name within a class, differentiated by their parameter lists (i.e., the number and/or type of parameters).
- It allows a method to perform different tasks based on the input parameters.
- In Python, method overloading is not supported in the same way as programming languages like Java or C++.
- However, you can achieve similar behaviour using default values for function parameters as one possible option.
- You can also use the *args and *kwargs concept to receive a varying parameter list.

Implementing Method Overloading

```
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
```

Poly: Duck Typing

- Duck typing is where the type or class of an object is less important than the methods or properties it possesses.

```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!
```

Let’s take a short break

CoGrammar

Let’s get coding!

CoGrammar

Questions and Answers

CoGrammar

Summary

- Why OOP is Essential in Programming
- Implementing a Class
- Usage of Access Control
- Principles of Encapsulation and Abstraction
  - Encapsulation bundles data and methods that operate on the data within a single unit (class), hiding details.
  - Abstraction focuses on representing the essential features of an object while hiding unnecessary details, improving code readability and maintenance.
- Demonstration of Inheritance and Polymorphism

Thank you for attending

SKILLS FOR LIFE
SKILLS BOOTCAMPS

Department for Education

CoGrammar