Welcome to this CoGrammar session:

Inheritance

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**

Skills Bootcamp
8-Week Progression Overview

Fulfil 4 Criteria to Graduation

✅ Criterion 1: Initial Requirements

- ***Timeframe:*** First 2 Weeks
- ***Guided Learning Hours (GLH):*** Minimum of 15 hours
- ***Task Completion:*** First four tasks

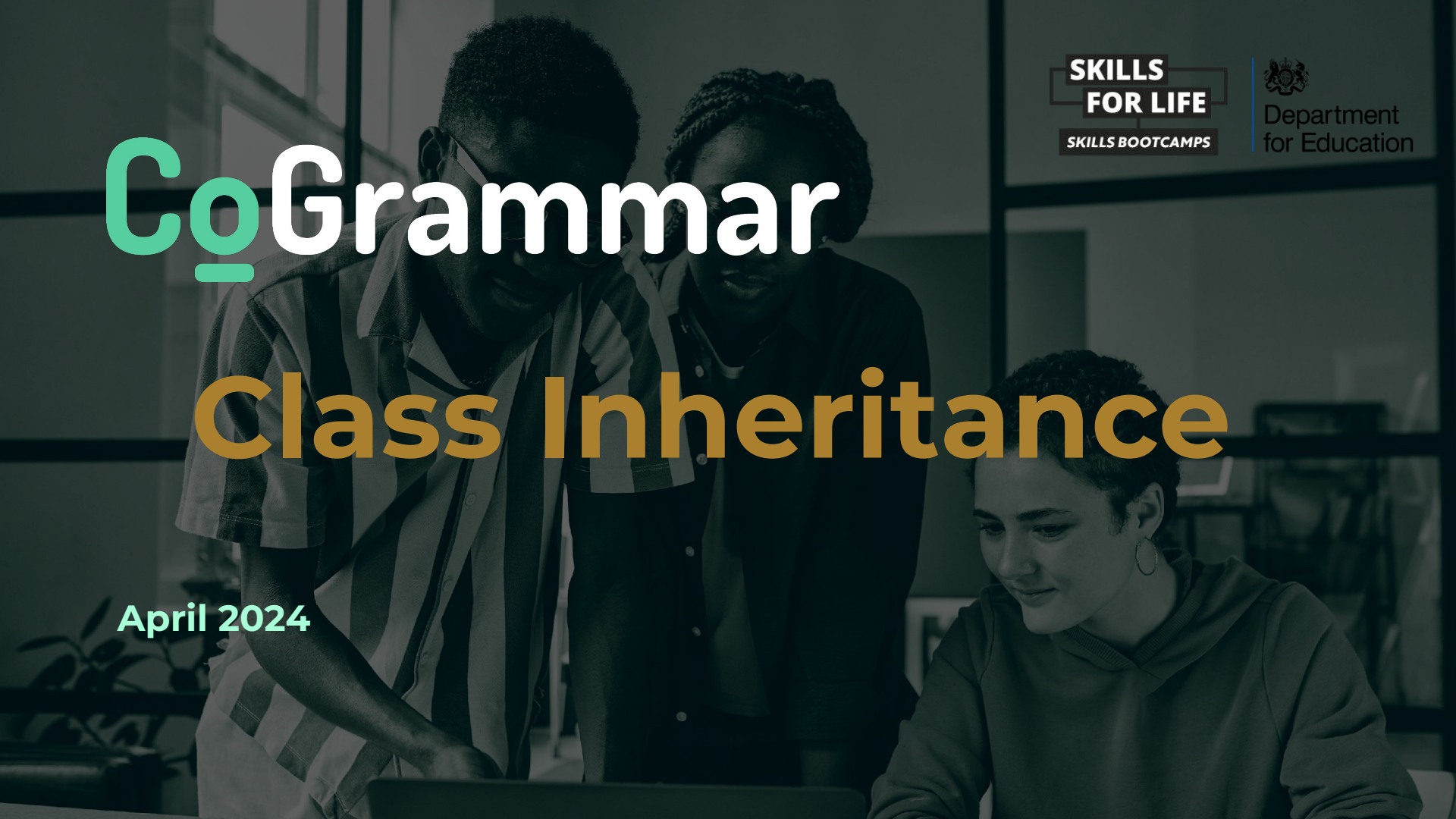
Due Date: 24 March 2024

✅ Criterion 2: Mid-Course Progress

- ***Guided Learning Hours (GLH):*** **60**
- ***Task Completion:*** 13 tasks

Due Date: 28 April 2024

CoGrammar



SKILLS FOR LIFE
SKILLS BOOTCAMPS

Department for Education

CoGrammar

Class Inheritance

April 2024

Learning Objectives

- Define inheritance and its role in object-oriented programming.
- Implement and utilise the principles of inheritance within your own projects.
- Use special methods in your classes to harness the full power of object-oriented programming in Python.
- Describe and utilise polymorphism with the use of method overriding and duck typing.

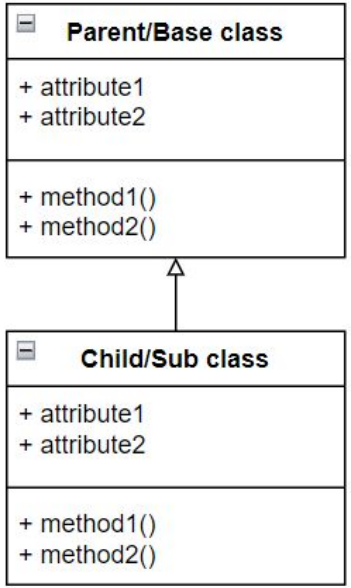
Inheritance

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.

What is Inheritance?



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.

Inheritance

- Here we have an example of having a child class inherit from a parent class

```python
# Base/Parent class
class Animal:
    def __init__(self, sound: str) -> None:
        self.sound = sound

    def make_sound(self) -> str:
        return f"The {type(self).__name__} goes {self.sound}"
```



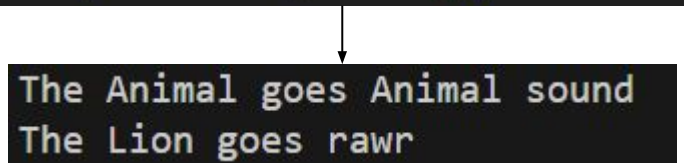
```python
# Sub/Child class
class Lion(Animal):
    pass
```

Inheritance

- Let's create one of each object and see how they behave.

```
animal = Animal("Animal sound")
lion = Lion("rawr")

print(animal.make_sound())
print(lion.make_sound())
```



```
The Animal goes Animal sound
The Lion goes rawr
```

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.

Method Overriding

- Let's override the make_sound() method.

```
# Sub/Child class
class Lion(Animal):
    pass
```

```
# Sub/Child class
class Lion(Animal):

    def make_sound(self) -> str:
        return f"The fierce lion let's out a big {self.sound}!!!!!!"
```

Method Overriding

- This will now be the new behaviour of the make_sound() method in the Lion class.

```
lion = Lion("rawr")
print(lion.make_sound())
```

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.

Methods overriding and Super()

- The method you will override the most will be __init__()
- When adding more instance attributes to the subclass we have to call to the base class __init__() to avoid having to redeclare all our base class attributes.
- We can use super().__init__() to call the constructor of the base class and set the values if the inherited attributes.

Methods overriding and Super()

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

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

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

Methods overriding and Super()

```
class BaseClass:
    # Base class definition
    def print_name(self):
        print(self.name)

class SubClass(BaseClass):
    # Subclass definition
    def print_name(self):
        print("Code before base method call.")
        super().print_name()
        print("Code after base method call.")
```

Multiple Inheritance

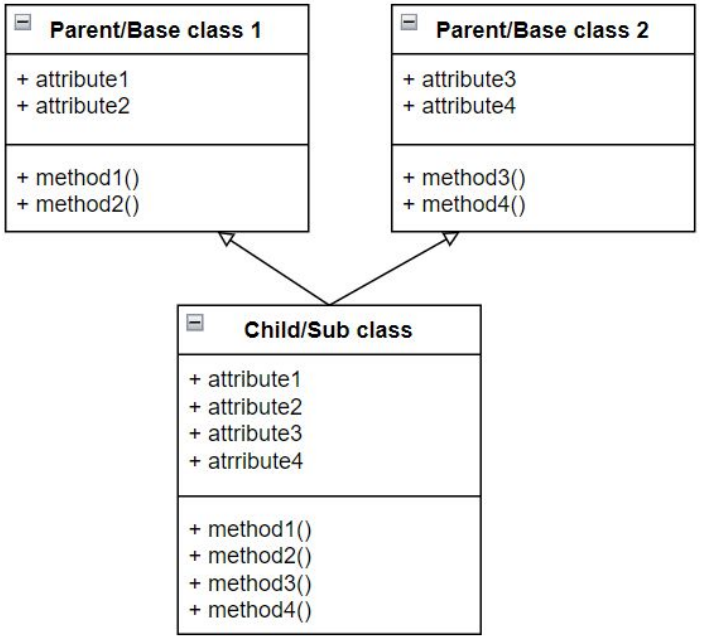
- Python allows multiple inheritance as well.
- This means we can have a subclass that inherits attributes and properties from more than one base class.

```
class BaseClass:
    # Base class definition
    pass

class BaseClassA:
    # Base class definition
    pass

class SubClass(BaseClass, BaseClassA):
    # Subclass definition
    pass
```

Multiple Inheritance



isinstance() and issubclass()

- We can determine if an object is an instance of a particular class using isinstance()
  - E.g. isinstance(object, ClassType)
- We can determine if a class is a subclass of another using issubclass()
  - E.g. issubclass(SubClass, BaseClass)

isinstance()

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

person = Person("Peter", "Parker")
print(isinstance(person, Person)) # True
```

issubclass()

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

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

print(issubclass(Student, Person)) # True
```

Special Methods

CoGrammar

__init__()

- The first special method you have seen and used is __init__().
- We use this method to initialize our instance variables and run any setup code when an object is being created.
- The method is automatically called when using the class constructor and the arguments for the method are the values given in the class constructor.

__init__()

```
class Student:

    def __init__(self, fullname, student_number):
        self.fullname = fullname
        self.student_number = student_number

new_student = Student("John McClane", "DH736648")
```

Objects As Strings

- You have probably noticed when using print() that some objects are represented differently than others.
- Some dictionaries and list have {} and [] in the representation and when we print an object we get a memory address <__main__.Person object at 0x000001EBCA11E650>
- We can set the string representations for our objects to whatever we like using either __repr__() or __str__()

__repr__()

- This method returns a string for an official representation of the object.
- __repr__() is usually used to build a representation that can assist developers when working with the class.
- The representation will contain extra information about the object that the user would not necessarily see.

__repr__()

```python
class Student:

    def __init__(self, fullname, student_number):
        self.fullname = fullname
        self.student_number = student_number

    def __repr__(self):
        return f"Student({self.fullname}, {self.student_number})"

new_student = Student("Percy Jackson", "PJ323423")
```

__str__()

- This method return a representation for your object when the str() function is called.
- When your object is used in the print function it will automatically try to cast your object to a string and will then receive the representation returned by __str__()
- This is usually a representation that users will see.

__str__()

```python
class Student:

    def __init__(self, fullname, student_number):
        self.fullname = fullname
        self.student_number = student_number

    def __str__(self):
        return f"Fullname:\t{self.fullname}\nStudent Num:\t{self.student_number}\n"

new_student = Student("Percy Jackson", "PJ323423")
print(new_student)
```

Special Methods And Math

- Special methods also allow us to set the behaviour for mathematical operations such as +, -, *, /, **
- Using these methods we can determine how the operators will be applied to our objects.
- E.g. When trying to add two of your objects, x and y, together python will try to invoke the __add__() special method that sits inside your object x. The code inside __add__() will then determine how your objects will be added together and returned.
- x + y -> x.__add__(y)

Special Methods And Math

```python
class MyNumber:

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

    def __add__(self, other):
        return MyNumber(self.value + other.value)

num1 = MyNumber(10)
num2 = MyNumber(5)
num3 = num1 + num2
print(num3.value) # Output: 15
```

Special Methods And Math

- Some mathematical special operators that are available are:
  - Add -> __add__(self, other)
  - Subtract -> __sub__(self, other)
  - Multiply -> __mul__(self, other)
  - Divide -> __truediv__(self, other)
  - Power -> __pow__(self, other)

Container-Like Objects

- Using special methods we can also incorporate behaviour that we see in container-like objects such as iterating, indexing, adding and removing items, and getting the length.
- E.g. When we try to get an item from a list the special method __getitem__(self,key) is called. We can then override the behaviour of the method to return the item we desire.
- Object[y] -> Object.__getitem__(y)

Container-Like Objects

```python
class ContactList:

    def __init__(self):
        self.contact_list = []

    def add_contact(self, contact):
        self.contact_list.append(contact)

    def __getitem__(self, key):
        return self.contact_list[key]


contact_list = ContactList()
contact_list.add_contact("Test Contact")
print(contact_list[0]) # Output: Test Contact
```

Container-Like Objects

- Some special methods to add for container-like objects are:
  - Length -> __len__(self)
  - Get Item -> __getitem__(self, key)
  - Set Item -> __setitem__(self, key, item)
  - Contains -> __contains__(self, item)
  - Iterator -> __iter__(self)
  - Next -> __next__(self)

Comparators

- The last special methods we will look at are comparators.
- We will use these methods to set the behaviour when we try to compare our objects to determine which one is smaller or larger or are they equal.
- E.g. When trying to see if object x is greater than object y. The method x__gt__(y) will be called to determine the result. We can then set the behaviour of __gt__() inside our class.
- x > y -> x.__gt__(y)

Comparators

```python
class Student:

    def __init__(self, fullname, student_number, average):
        self.fullname = fullname
        self.student_number = student_number
        self.average = average

    def __gt__(self, other):
        return self.average > other.average

student1 = Student("Peter Parker", "PP734624", 88)
student2 = Student("Tony Stark", "TS23425", 85)
print(student1 > student2) # Output: True
```

Polymorphism

CoGrammar

Method Overriding

- When extending or changing behaviour of a parent class is best to make sure we do it in a polymorphic way. Here we changed the behaviour of the make_sound method in the Lion class but we can still use it in our animal_make_sound() function.

```python
class Animal:

    def __init__(self) -> None:
        self.sound = ""

    def make_sound(self):
        return f"The animal goes {self.sound}"
```

```python
class Lion(Animal):

    def __init__(self) -> None:
        super().__init__("Rawr")

    def make_sound(self):
        return f"The lion goes {self.sound}"
```

```python
def animal_make_sound(animal):
    print(animal.make_sound())
```

CoGrammar

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!
```

Questions and Answers

CoGrammar

Let’s take a short break

CoGrammar

Summary

CoGrammar

Questions and Answers

CoGrammar

Summary

1. Inheritance
   - ➢ Allows us to reuse code and reduce redundancy by inheriting attributes and properties from a parent.
2. Special Methods
   - ➢ Also called dunder or magic methods are used to implement special behaviours into our classes to allow them to interact with built in python methods.
3. Polymorphism
   - ➢ An idea where we care about the behaviours of an object rather than the pacific type of the object.

Thank you for attending

SKILLS FOR LIFE
SKILLS BOOTCAMPS

Department for Education

CoGrammar