Welcome to this CoGrammar Tutorial:
Class Inheritance
and
Magic Methods

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** throughout this session, should you wish to ask any follow-up questions.
- 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**
- If you are hearing impaired, please kindly use your computer's function through Google chrome to enable captions.

Safeguarding & Welfare

We are committed to all our students and staff feeling safe and happy; we want to make sure there is always someone you can turn to if you are worried about anything.

If you are feeling upset or unsafe, are worried about a friend, student or family member, or you feel like something isn't right, speak to our safeguarding team:

Ian Wyles
Designated Safeguarding Lead

Simone Botes

Nurhaan Snyman

Rafiq Manan

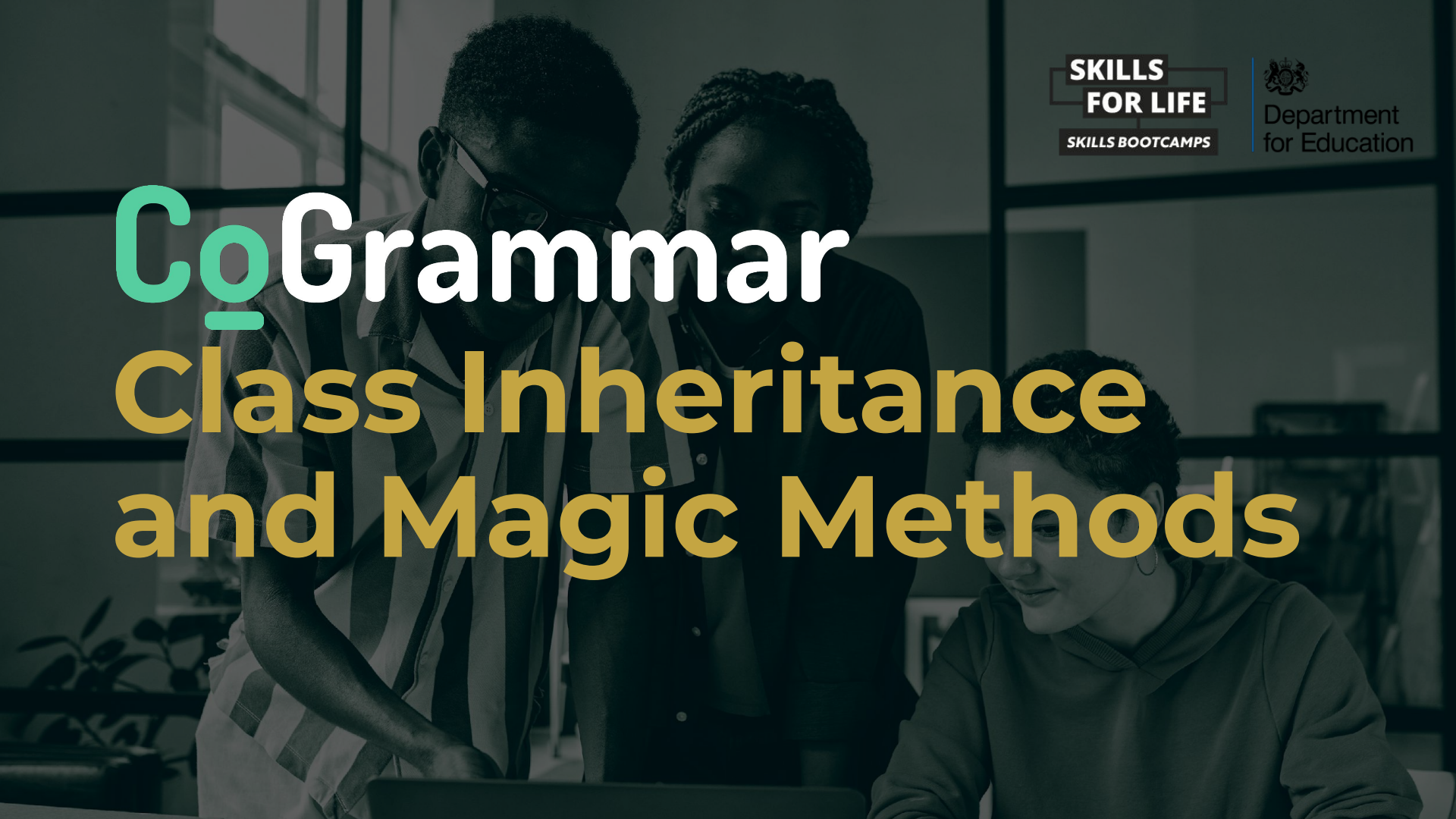
Ronald Munodawafa

Tevin Pitts

Scan to report a safeguarding concern

or email the Designated Safeguarding Lead:
Ian Wyles
safeguarding@hyperiondev.com

CoGrammar HyperionDev


SKILLS
FOR LIFE
SKILLS BOOTCAMPS
Department
for Education
CoGrammar
Class Inheritance
and Magic Methods

Poll

What will the following code output when using method overriding and `super()`?

```
1   class A:
2       def show(self):
3           return "Class A"
4
5   class B(A):
6       def show(self):
7           return super().show() + " and Class B"
8
9   b = B()
10  print(b.show())
```

A. Error: super() cannot be used here
B. Class B
C. Class A and Class B

Poll

What is the output of the following code demonstrating magic methods and operator overloading?

```
1   class Vector:
2       def __init__(self, x, y):
3           self.x = x
4           self.y = y
5
6       def __add__(self, other):
7           return Vector(self.x + other.x, self.y + other.y)
8
9       def __str__(self):
10          return f"({self.x}, {self.y})"
11
12  v1 = Vector(1, 2)
13  v2 = Vector(3, 4)
14  v3 = v1 + v2
15  print(v3)
```

a. (1, 2)

b. (4, 6)

c. Error: + operator not supported

Learning Objectives & Outcomes

- Define and implement inheritance in Python classes.
- Apply method overriding to customise inherited methods.
- Use multiple inheritance to create complex class structures.
- Utilise magic methods for custom behaviour and operator overloading.
- Develop Python programs incorporating inheritance and special methods effectively.

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.
- By 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/Super class**
  - The parent or base class contains all the attributes and properties we want to inherit.
- **Child/Subclass/Derived class**
  - The child or sub class will inherit all the attributes and properties of the parent class.

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, in the subclass.
- 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 super().__init__() from the Person class to set the values for the attributes “name” and “age”.

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

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

Multiple Inheritance

```python
class Teacher:
    def teach(self):
        return "Teaching"

class Researcher:
    def research(self):
        return "Conducting research"

class Professor(Teacher, Researcher):
    pass

# Create a Professor object
prof = Professor()

# Call methods from both parent classes
print(prof.teach())     # Output: Teaching
print(prof.research()) # Output: Conducting research
```

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

Special Methods

CoGrammar

Instantiation: `__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.

Representation: Objects As Strings

```
class Student:
    def __init__(self, fullname, student_number):
        # Initialize instance variables
        self.fullname = fullname          # Set the full name of the student
        self.student_number = student_number  # Set the student number

# Create a Student object with specific values
student_1 = Student("Jacob", "ABCD1234")

# Print the student object
print(student_1)
```

`__str__()` or `__repr__()`

- You've likely noticed that some objects display differently when using `print()`.
- Dictionaries use `{}`, lists use `[]`, and printing an object often shows a memory address like `<__main__.Person object at 0x000001EBCA11E650>`.
- We can customize how our objects are represented by using the `__repr__()` or `__str__()` methods.

`__str__()`

- The `__str__()` method provides a string representation of an object when called.
- When an object is used with the `print()` function, Python automatically converts it to a string using the `__str__()` method.
- This string representation is generally intended for user display.

__str__()

```python
class Student:
    def __init__(self, fullname, student_number):
        # Initialize instance variables
        self.fullname = fullname            # Set the full name of the student
        self.student_number = student_number  # Set the student number

    def __str__(self):
        # Return a string representation of the object
        return f"Student: {self.fullname}, Number: {self.student_number}"

# Create a Student object with specific values
student_1 = Student("Jacob", "ABCD1234")

# Print the student object
print(student_1)
```

Operator Overloading: 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.

`__add__()`

- **E.g.**
  - When adding `x` and `y`, Python calls the `__add__()` method in `x`.
  - `__add__()` defines how the objects are added and returns the result.

Operator Overloading: Example

```
class Number:
    def __init__(self, value):
        self.value = value

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

    def __str__(self):
        return str(self.value)

# Create two objects
x = Number(10)
y = Number(5)

# Add the two objects using +
result = x + y

# Print the result
print(result)   # Output: 15
```

Comparator Special Methods

- Define object **comparison** behavior
- Used for determining relative **size** or **equality**
- Examples:
  - **x > y** calls **x.**`__gt__(y)`
  - **x < y** calls **x.**`__lt__(y)`
  - **x == y** calls **x.**`__eq__(y)`
- Customizing these **methods** controls comparison **outcomes**

Comparators: Example

```python
class Student:
    def __init__(self, fullname, student_number, average):
        # Initialize instance variables
        self.fullname = fullname              # Set the full name of the student
        self.student_number = student_number  # Set the student number
        self.average = average                # Set the average mark of the student

    def __gt__(self, other):
        # Define behavior for the 'greater than' comparison
        return self.average > other.average

# Create two Student objects with specific values
student_1 = Student("Jacob", "ABCD1234", 95)
student_2 = Student("Yrneh", "ABCD1235", 90)

# Compare students based on their average marks
print(student_1 > student_2)
```

Addressing 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.
- Code: Object[y] ⟶ Executes: Object.`__getitem__(y)`

Addressing Container-Like Objects

```
class CustomContainer:
    def __init__(self, items):
        self.items = items # Initialize with a list of items

    def __getitem__(self, index): # Customize behavior for indexing
        if index < 0 or index >= len(self.items):
            raise IndexError("Index out of range")
        return self.items[index]

# Create an instance of CustomContainer with a list of items
container = CustomContainer(['apple', 'banana'])
# Access items using indexing
print(container[0])  # Output: apple
print(container[1])  # Output: banana
# Uncommenting the following line will raise an IndexError
# print(container[3])  # IndexError: Index out of range
```

Special Methods Addressing 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)`

Lesson Conclusion and Recap

Recap the key concepts and techniques covered during the lesson.

- **Inheritance** allows a subclass to inherit attributes and methods from a superclass, enabling code reuse and structured organisation.
- **Superclass and Subclass**: The superclass (parent) provides the inherited properties, while the subclass (child) extends or modifies them.
- **Method Overriding**: Subclasses can override inherited methods to provide specific implementations, allowing customization.
- **`super()`**: The `super()` function allows subclasses to call methods from the superclass, often used in constructors or overridden methods.
- **Benefits**: Inheritance simplifies code by reusing functionality, enhancing extensibility, and maintaining a clear hierarchy.

Let’s get coding!

CoGrammar

Questions and Answers

CoGrammar

Thank you for attending

SKILLS FOR LIFE
SKILLS BOOTCAMPS

Department for Education

CoGrammar