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.

CoGrammar

SKILLS FOR LIFE
SKILLS BOOTCAMPS

Department for Education

CoGrammar

Class Inheritance and Magic Methods

September 2024

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**

- We would love your **feedback** on lectures: **Feedback on Lectures**

Safeguarding & Welfare

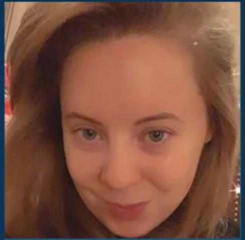
We are committed to all our students and staff feeling safe and happy; we want to make sure there is always someone you can turn to if you are worried about anything.

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

Rafiq Manan



Charlotte Witcher

Nurhaan Snyman

Ronald Munodawafa

Tevin Pitts

Scan to report a safeguarding concern

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

CoGrammar HyperionDev

Skills Bootcamp Progression Overview

To be eligible for a certificate of completion, students must fulfil three specific criteria. These criteria ensure a high standard of achievement and alignment with the requirements for the successful completion of a Skills Bootcamp.

✅ Criterion 1 - Meeting Initial Requirements

Criterion 1 involves specific achievements within the first two weeks of the program. To meet this criterion, students need to:

- Attend a minimum of 7-8 hours per week of guided learning (lectures, workshops, or mentor calls) within the initial two-week period, for a total minimum of 15 guided learning hours (GLH), by no later than 15 September 2024.
- Successfully complete the Initial Assessment by the end of the first 14 days, by no later than 15 September 2024.

Skills Bootcamp Progression Overview

✅ Criterion 2 - Demonstrating Mid-Course Progress

Criterion 2 involves demonstrating meaningful progress through the successful completion of tasks within the first half of the bootcamp.
To meet this criterion, students should:

- Complete 42 guided learning hours and the first half of the assigned tasks by the end of week 7, no later than 20 October 2024.

CoGrammar

Skills Bootcamp Progression Overview

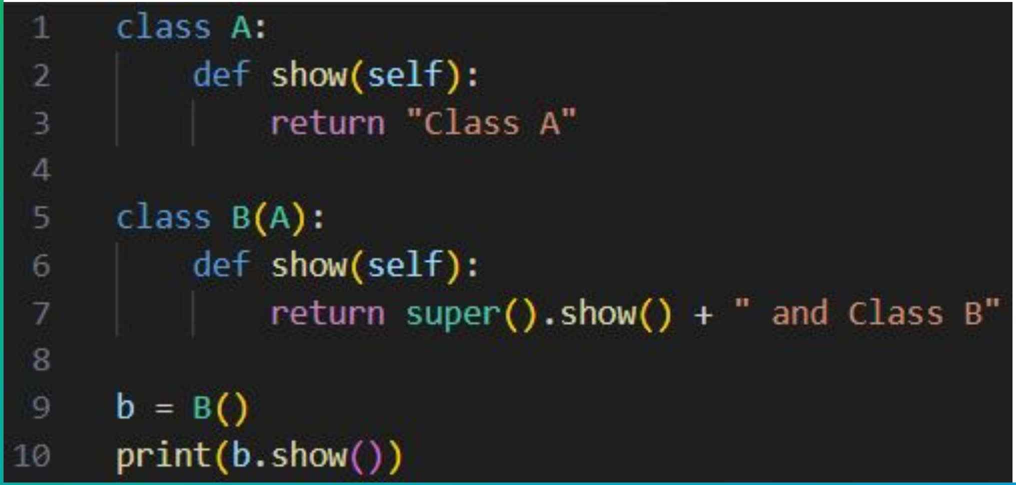
✅ Criterion 3 - Demonstrating Post-Course Progress

Criterion 3 involves showcasing students' progress after completing the course. To meet this criterion, students should:

- Complete all mandatory tasks before the bootcamp's end date. This includes any necessary resubmissions, no later than 22 December 2024.
- Achieve at least 84 guided learning hours by the end of the bootcamp, 22 December 2024.

Poll

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



```python
class A:
    def show(self):
        return "Class A"

class B(A):
    def show(self):
        return super().show() + " and Class B"

b = B()
print(b.show())
```

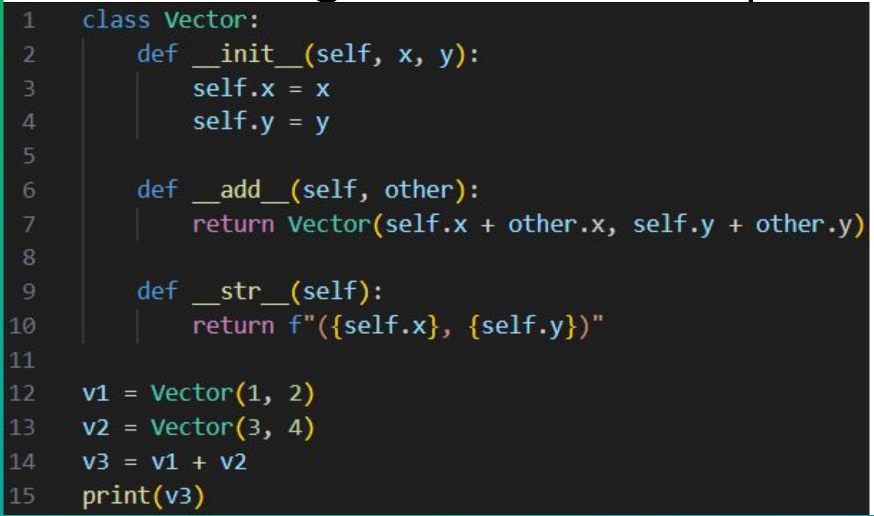
A. Error: super() cannot be used here

B. Class B

C. Class A and Class B

Poll

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



```python
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __str__(self):
        return f"({self.x}, {self.y})"

v1 = Vector(1, 2)
v2 = Vector(3, 4)
v3 = v1 + v2
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

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.

CoGrammar

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.

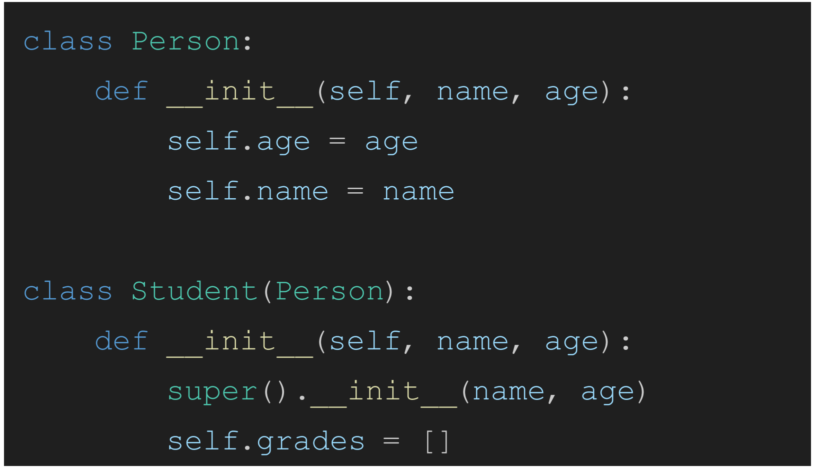
super()

- 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.

CoGrammar

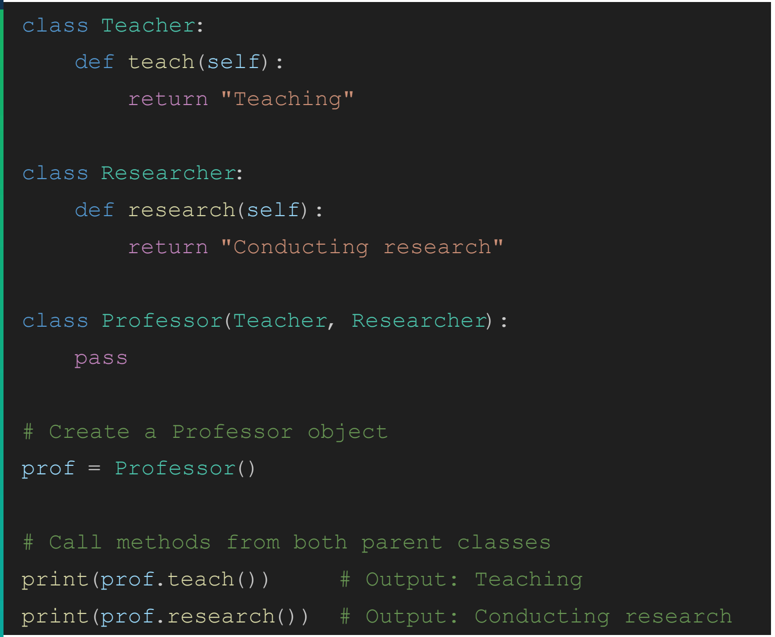
Method overriding and super()

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.

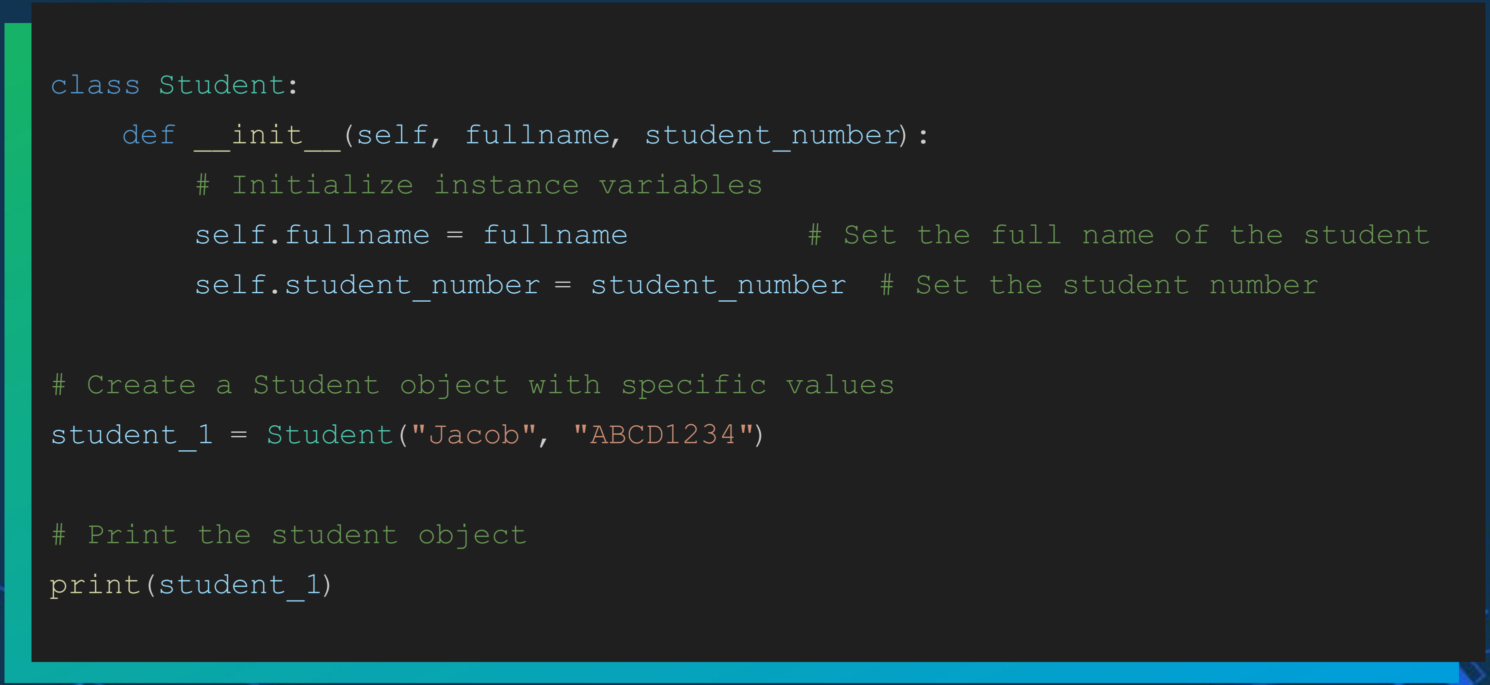
Special Methods

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



```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

# 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>`.

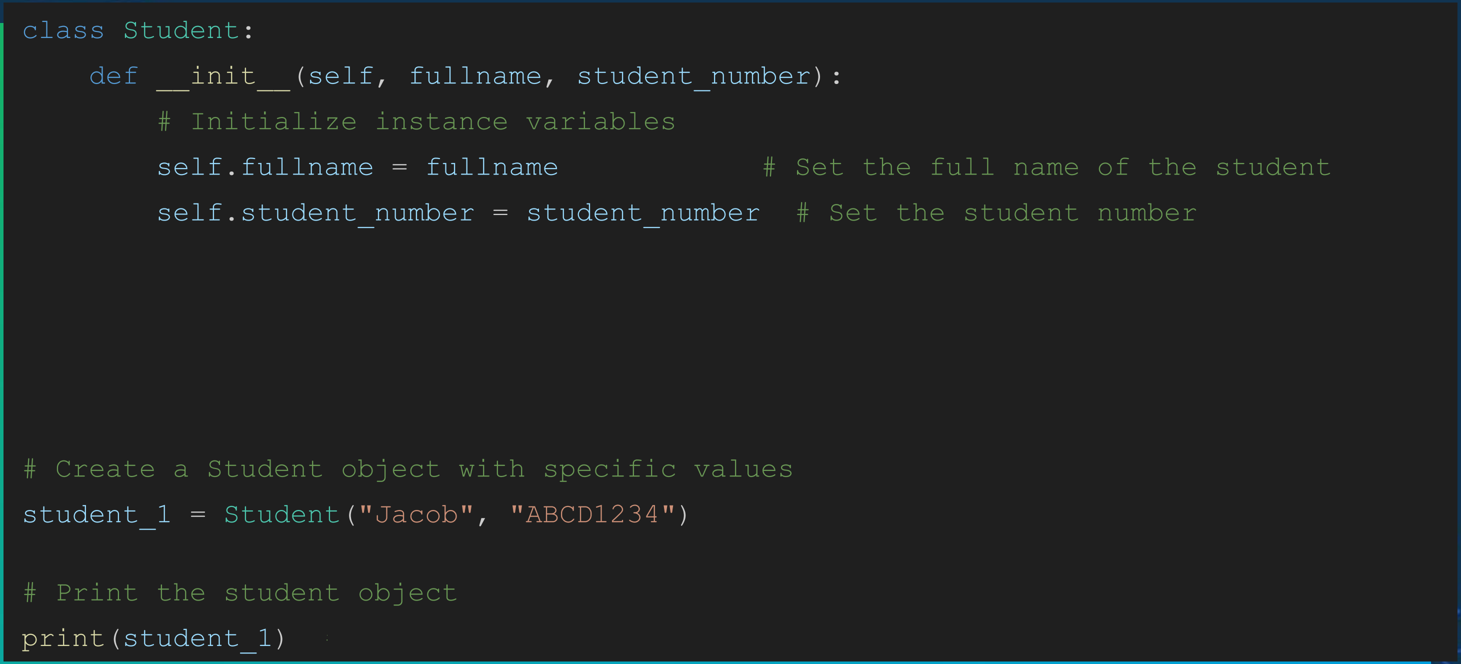
- We can customize how our objects are represented by using the `__repr__()` or `__str__()` methods.

`__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.

CoGrammar

__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




# 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.

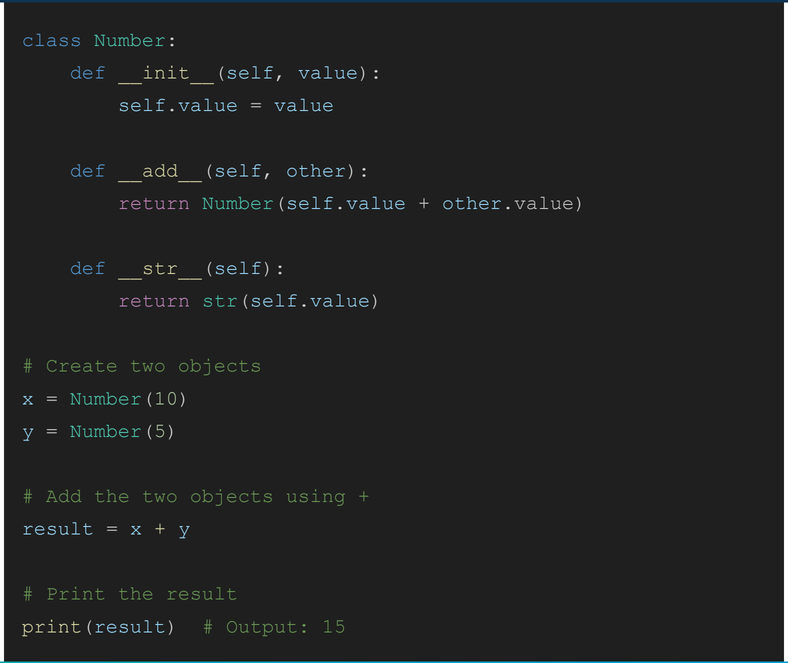
CoGrammar

__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.

CoGrammar

Operator Overloading: Example

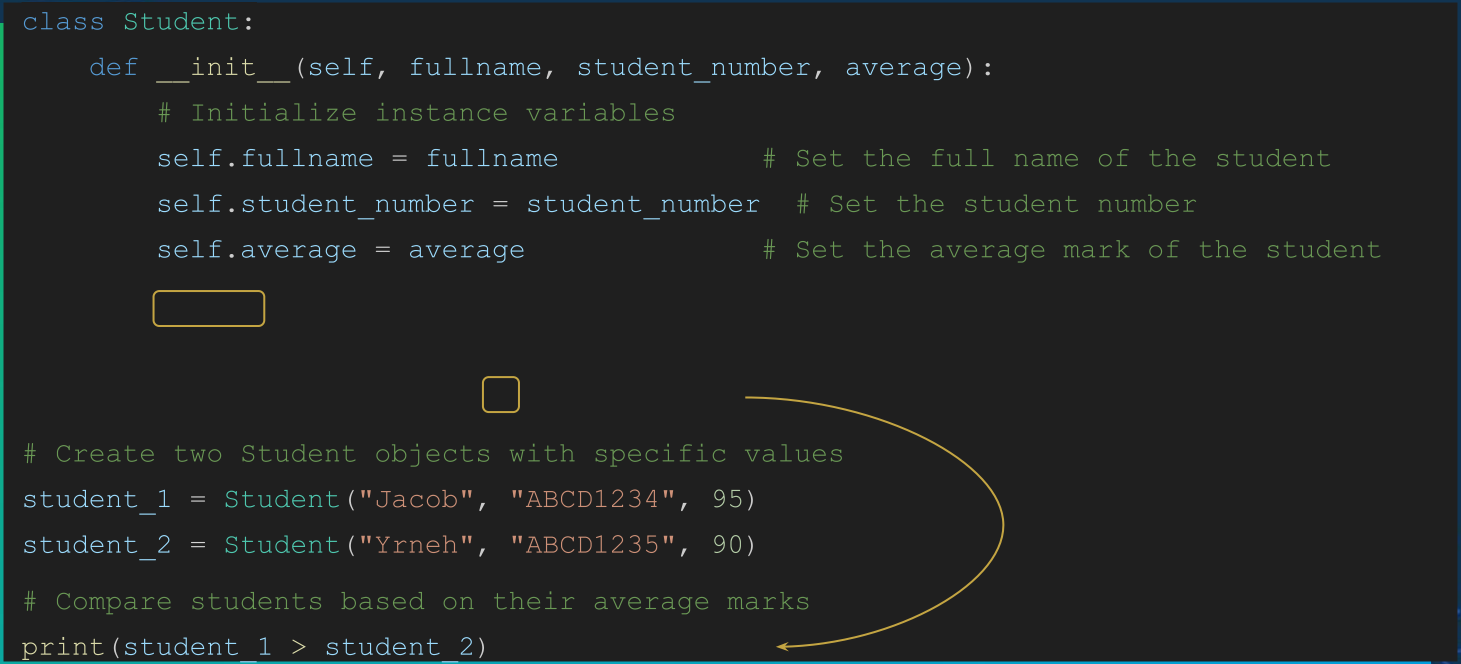


```python
class Number:
    def __init__(self, value):
        self.value = value

    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



# 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)
```

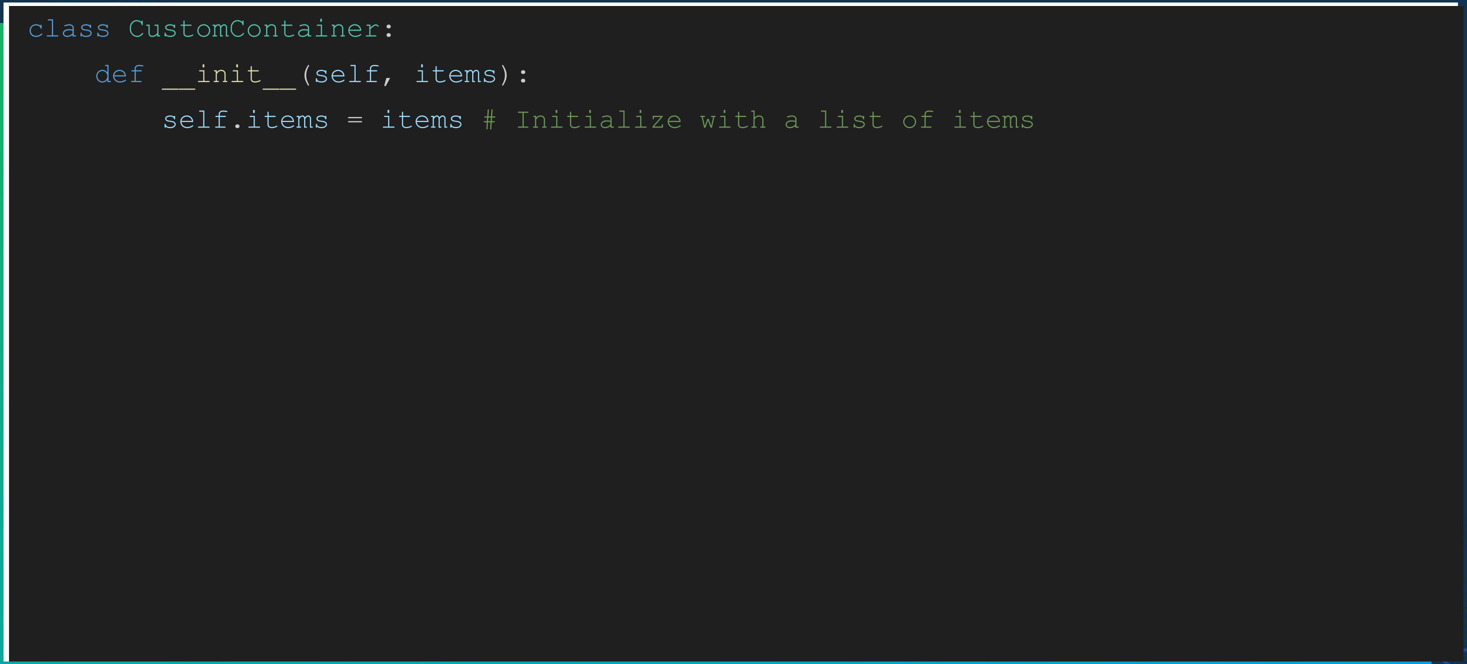
Addressing Container-Like Objects

- 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
```

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!

Questions and Answers

CoGrammar

Thank you for attending

SKILLS FOR LIFE
SKILLS BOOTCAMPS

Department for Education

CoGrammar