Welcome to this CoGrammar lecture:

Classes III – Special Methods

The session will start shortly...

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

Software Engineering Session Housekeeping

- For all **non-academic questions**, please submit a query: www.hyperiondev.com/support
- We would love your **feedback** on lectures: Feedback on Lectures

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 throughout the 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

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

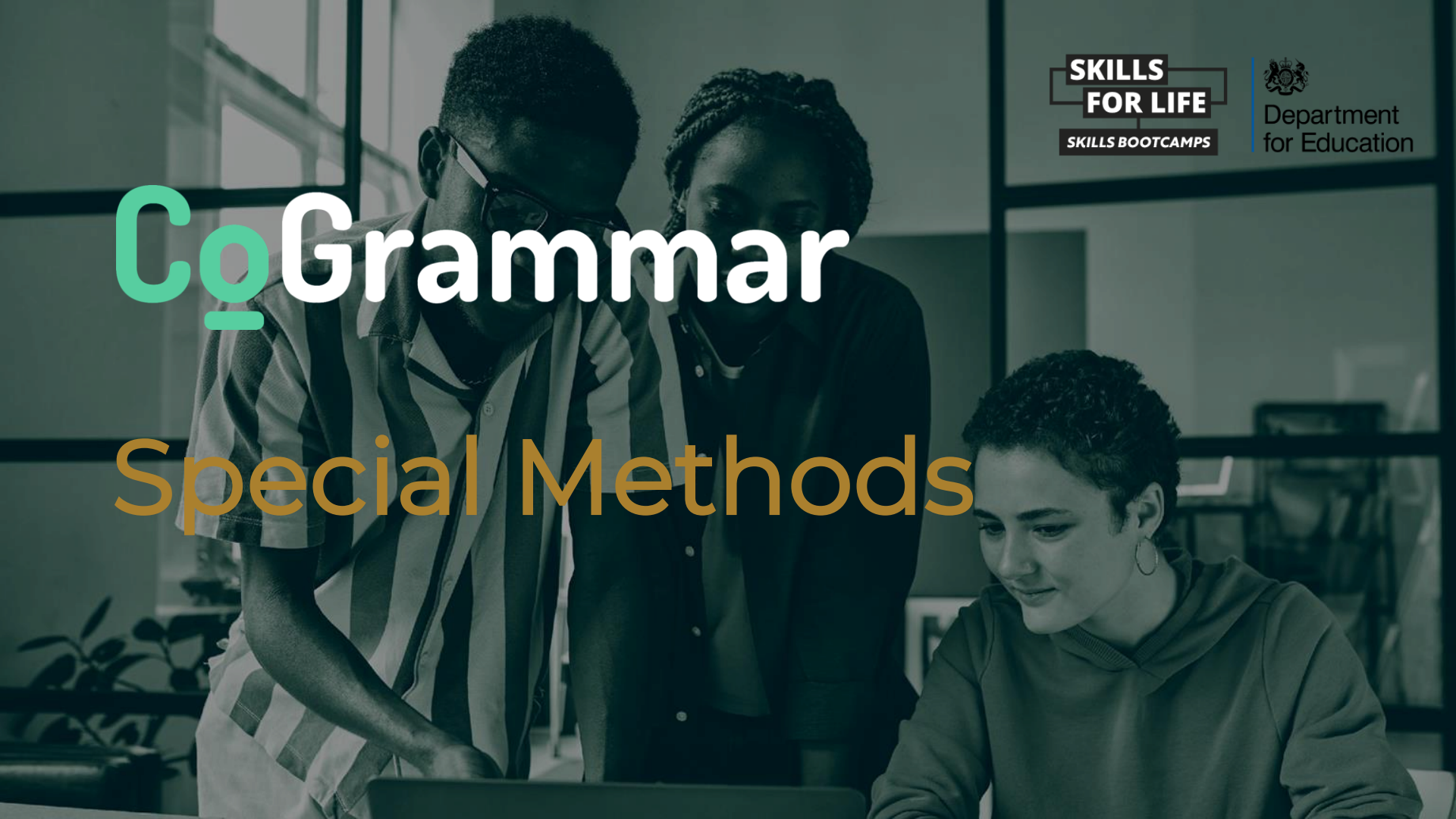
Skills Bootcamp Progression Overview

✓ 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.

CoGrammar


SKILLS
FOR LIFE
SKILLS BOOTCAMPS
Department
for Education
CoGrammar
Special Methods

Poll

1. **How familiar are you with the concept of special methods (dunder methods) in Python?**
   A. Very familiar; I have used them in my projects.
   B. Somewhat familiar; I know they exist but haven't used them much.
   C. Not familiar; I have not heard of them before.

Poll

2. Which of the following special methods have you used or encountered in your coding experience? (Select all that apply)

A. __init__ (Constructor)

B. __str__ (String representation)

C. __repr__ (Official string representation)

D. __eq__ (Equality comparison)

E. __lt__ (Less than comparison)

F. None of the above.

Learning Outcomes

- Remember the purpose of special methods in Python.
- Explain how special methods (like __init__, __str__, __repr__) enhance object-oriented programming.
- Apply special methods to create well-structured Python classes.
- Analyse how different special methods influence the behaviour of Python objects.
- Evaluate when and why specific special methods should be used in software design.

Learning Outcomes

- Create a Python class that implements at least three special methods.
- Describe and utilise polymorphism with the use of method overriding and duck typing.

Special Methods

CoGrammar

What are Special Methods?

- Special methods in Python are predefined methods that allow developers to define how objects of a class should behave in certain situations.
- Also known as magic methods or dunder methods (short for "double underscore") because they begin and end with double underscores, __
- These methods allow custom objects to integrate seamlessly with Python's built-in features, such as string representations, arithmetic operations, comparisons, and more.

Constructors and Destructor

CoGrammar

__init__()

- The first special method you have seen and used is __init__().
- We use this method to initialise 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")
```

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.

Destructor - Example

```python
class FileManager:
    def __init__(self, filename):
        self.file = open(filename, 'w')
        print(f"Opened {filename}.")

    def __del__(self):
        self.file.close()
        print("File closed.")

# Create an instance and write to the file
file_manager = FileManager("example.txt")
file_manager.file.write("Hello, World!")

# Explicitly delete the object to trigger the destructor
del file_manager
```

Objects as Strings

CoGrammar

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.
- This representation will contain extra information in the method about the object that that is not meant for the user.

__repr__()

```python
class Student:

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

    def __str__(self):
        # Including memory address and internal state, useful for debugging
        return (f"<Student(name={self.full_name!r}, "
                f"S_Number={self.student_number!r}, id={hex(id(self))})>")


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

print(new_student)
# Output: <Student(name='Percy Jackson', S_Number='PJ323423', id=0xc303747f50)>
```

__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 for users to see.

__str__()

```python
class Student:

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

    def __str__(self):
        return (f"Full Name:\t{self.full_name}\n"
                f"Student Num: \t{self.student_number}")


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

print(new_student)
# Output:   Full Name:      Percy Jackson
#           Student Num:    PJ323423
```

Container-Like Objects

CoGrammar

Container-Like Objects

- A container-like object is any object that can hold or store other objects. These objects allow you to group multiple items together and already provide various methods for accessing, adding, removing, and iterating over these items.
- Using special methods we can also incorporate the behaviour that we see in container-like objects.
- 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 default behaviour of the method to return the result we desire.

Key Characteristics

- Holds Multiple Items: Container objects can store more than one value, often of various types, in a single entity.
- Supports Iteration: They can be iterated over, allowing you to loop through their contents easily.
- Dynamic Sizing: Many container-like objects can grow and shrink in size as items are added or removed.
- Indexing and Slicing: Some containers support accessing items using indices or slicing.
- We want our custom objects to mimic this behaviour.

Implementing Container-Like Behaviour

```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:
  - len(object) -> __len__(self)
  - object[key]-> __getitem__(self, key)
  - object[key] = item -> __setitem__(self, key, item)
  - item in object -> __contains__(self, item)
  - variable = object(parameter) -> __call__(self, parameter)
  - iter(object) or ‘for item in object’ -> __iter__(self)
  - next(iterator) -> __next__(self)

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__)
```

Comparators

CoGrammar

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
```

Other Comparators

- Commonly Used Special Methods for Comparison:
  - __eq__(self, other): Behaviour for equality (==)
  - __ne__(self, other): Behaviour for inequality (!=)
  - __lt__(self, other): Behaviour for less-than (<)
  - __le__(self, other): Behaviour for less-than-or-equal (<=)
  - __gt__(self, other): Behaviour for greater-than (>)
  - __ge__(self, other): Behaviour for greater-than-or-equal (>=)

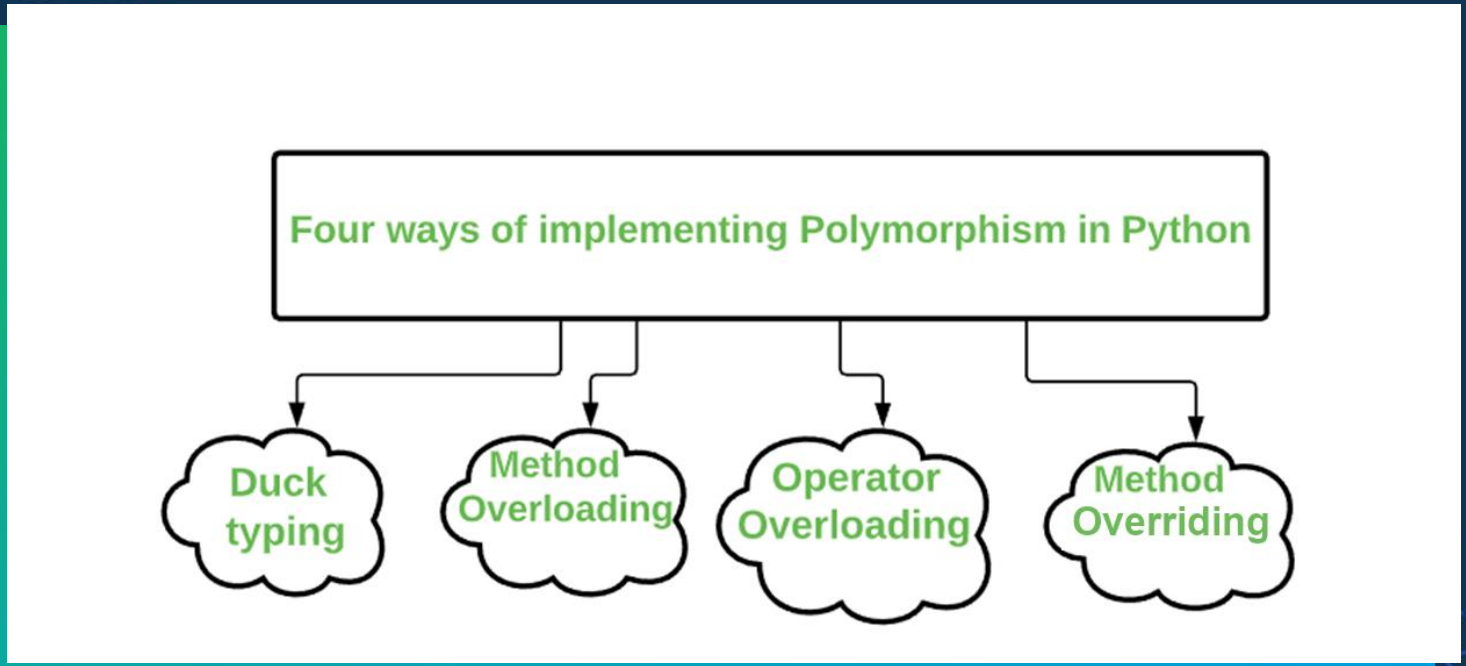
Polymorphism

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



Method Overriding

CoGrammar

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.
- When changing behaviour of a parent class, it is best to make sure we do it in a polymorphic way. Let's change the behaviour of the make_sound method in the Lion class to still use the method of the parent in our animal_make_sound() function.

Method Overriding...

```python
class Animal:       # Parent class
    def make_sound(self):
        return "Some generic animal sound"

class Lion(Animal): # Child class (Lion) overriding the make_sound method
    def make_sound(self):
        return "Roar"

# A function that uses the polymorphic behaviour of the make_sound method
def animal_make_sound(animal):
    print(animal.make_sound())

# Creating instances of Animal and Lion
generic_animal = Animal()
lion = Lion()

# Calling the function with both the parent and child class
animal_make_sound(generic_animal)  # Output: Some generic animal sound
animal_make_sound(lion)            # Output: Roar
```

Operator Overloading

CoGrammar

Poly: Operator Overloading

- Special methods 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)

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.
  - __pow__(self, other): Behaviour for the (**) operator.
  - __truediv__(self, other): Behaviour for the (/) operator.
  - __eq__(self, other): Behaviour for the (==) operator.

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
```

Method Overloading

CoGrammar

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
```

Duck Typing

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!
```

Let’s take a short break

CoGrammar

Demo Time!

CoGrammar

Poll

1. **What are special methods in Python primarily used for?**
   A. Creating user interfaces
   B. Defining custom behaviour for built-in operations
   C. Managing file I/O
   D. Optimising performance

Poll

2. **Which of the following special methods is called when an object is created?**

   A. __init__

   B. __str__

   C. __repr__

   D. __call__

CoGrammar

Poll

3. **Which special method is used to define a user friendly string representation of an object?**

   A. __repr__

   B. __str__

   C. __init__

   D. __eq__

Poll

4. **Which special methods do you feel you can effectively use in your programming now? (Select all that apply)**
   A. __init__ (Constructor)
   B. __str__ (String representation)
   C. __repr__ (Official string representation)
   D. __eq__ (Equality comparison)
   E. __lt__ (Less than comparison)

Poll

5. **How would you rate the overall effectiveness of the lesson on special methods?**
   A. Excellent; I learned a lot.
   B. Good; I learned some useful information.
   C. Fair; I didn't find it very helpful.
   D. Poor; I did not gain any understanding from the lesson.

Conclusion and Recap

CoGrammar

Conclusion and Recap

- Special Methods
  - Also called dunder or magic methods and are used to implement special behaviours into our classes to allow them to interact with built-in python methods.
- Polymorphism
  - An idea where we care about the behaviours of an object rather than the specific type of the object.
  - Can be applied with Method Overriding, Operator Overloading, Method Overloading and Duck Typing.

Learner Challenge – Option 1

- *Create a custom Stack class that implements special methods to behave like a stack data structure:*

1. Implement Special Methods:

- __init__(self): Initialise an empty stack.
- __len__(self): Return the number of items in the stack.
- __getitem__(self, index): Allow access to elements in the stack using an index.
- __str__(self): Return a string representation of the stack (e.g., "Stack: [3, 2, 1]").

2. Additional Methods:

- push(self, item): Add an item to the top of the stack.
- pop(self): Remove and return the item from the top of the stack. If the stack is empty, raise an IndexError.

Learner Challenge – Option 1

Questions to Reflect:

- How did implementing the __len__ and __getitem__ methods enhance the usability of your stack? Can you think of other methods you might add for further functionality?
- Are there any parts of your code that could be made more efficient or cleaner? If so, what changes would you make?

Learner Challenge – Option 2

- *Create a custom Fraction class that supports basic arithmetic operations and comparisons using special methods:*:

1. Special Methods to Implement:

- __init__(self, numerator, denominator): Initialise a fraction with a numerator and denominator.
- __add__(self, other): Implement addition of two fractions.
- __sub__(self, other): Implement subtraction of two fractions.
- __mul__(self, other): Implement multiplication of two fractions.
- __truediv__(self, other): Implement division of two fractions.
- __str__(self): Return a string representation of the fraction (e.g., "3/4").
- __eq__(self, other): Implement equality comparison between two fractions.

Learner Challenge – Option 2

Questions to Reflect:

- Did you consider simplifying fractions (e.g., reducing 4/8 to 1/2) in your implementation? Why might this be an important feature?
- How does override the __eq__ method enhance the functionality of your Fraction class?

Tasks To Complete

- T11 - OOP - Inheritance

Questions and Answers

CoGrammar

Thank you for attending

SKILLS FOR LIFE
SKILLS BOOTCAMPS

Department for Education

CoGrammar