CoGrammar

Week 17: Open Class 2

SKILLS FOR LIFE
SKILLS BOOTCAMPS

Department for Education

Software Engineering Lecture Housekeeping

- The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly. **(FBV: Mutual Respect.)**
- 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 Open Classes. You can submit these questions here: **Open Class Questions**

Software Engineering Lecture 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**

Lecture Objectives

1. Describe special methods and their use when working with classes and objects in Python.

2. Experiment with different special methods in your classes to see how they will add and change behaviour of your class.

__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 lists have {} and [] in the representation and when we print an objects 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__()

```
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__()

```
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__(a, y)

Special Methods And Math

```
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 also 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
```

*

Wrapping Up

Special Methods

We can use special methods to add and set specific behaviour for built-in python functions. We can add behaviour for string representation, mathematical operations, container-like objects and many more.

CoGrammar

Questions around special methods

CoGrammar

Thank you for joining