Welcome to this CoGrammar Tutorial: Class Inheritance and Multi-Dimensional Lists

The session will start shortly...

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

CoGrammar

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

- ***Guided Learning Hours (GLH):*** Minimum of 15 hours
- ***Task Completion:*** First 4 tasks

Due Date: 24 March 2024

✓ Criterion 2: Mid-Course Progress

- ***Guided Learning Hours (GLH):*** Minimum of 60 hours
- ***Task Completion:*** *First* 13 tasks

Due Date: 28 April 2024

CoGrammar

Skills Bootcamp Progression Overview

✓ Criterion 3: Course Progress

- ***Completion:*** All mandatory tasks, including Build Your Brand and resubmissions by study period end
- ***Interview Invitation:*** Within 4 weeks post-course
- ***Guided Learning Hours:*** Minimum of 112 hours by support end date (10.5 hours average, each week)

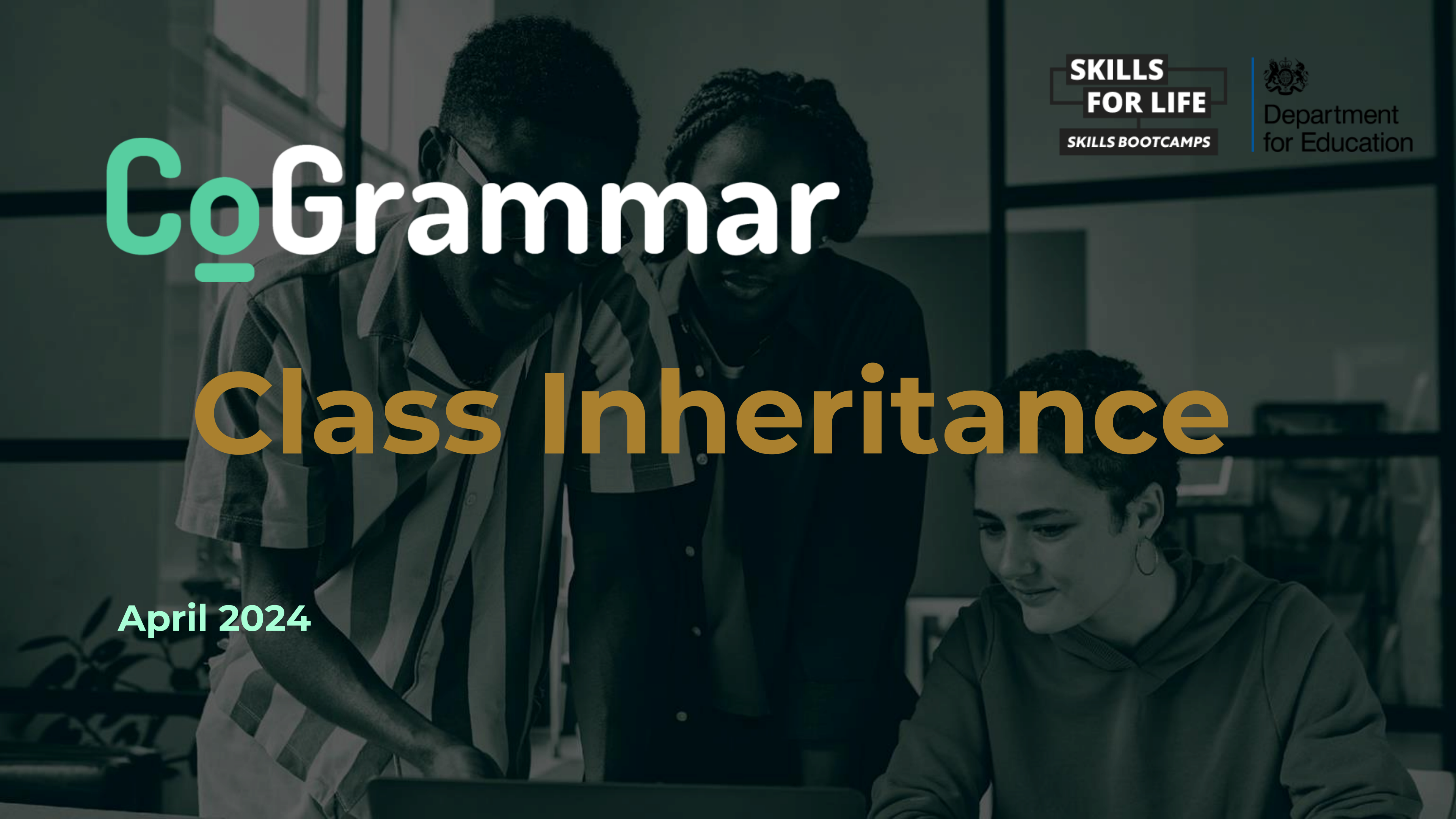
✓ Criterion 4: Demonstrating Employability

- ***Final Job or Apprenticeship Outcome:*** Document within 12 weeks post-graduation
- ***Relevance:*** Progression to employment or related opportunity

CoGrammar

Learning Objectives & Outcomes

- Implement and utilise the principles of inheritance within projects
- Implement multiple inheritances
- Utilise method overriding
- Incorporate special methods in classes including operator overloading
- Implement the different lists operations
- Implement 1D and higher dimensional lists


SKILLS
FOR LIFE
SKILLS BOOTCAMPS
Department
for Education
CoGrammar
Class Inheritance
April 2024

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.

CoGrammar

Inheritance...

- **Parent/Base class**
  - The parent or base class contains all the attributes and properties we want to inherit.
- **Child/Subclass**
  - 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.

- 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

Methods overriding and Super()

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

```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 = []
```

Methods overriding and Super()

```python
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.

```python
class BaseClass:
    # Base class definition
    pass

class BaseClassA:
    # Base class definition
    pass

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

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.

__init__()

```python
class Student:

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

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

CoGrammar

Representation: 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 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__()

CoGrammar

__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)
```

CoGrammar

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.

- 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.

CoGrammar

Operator Overloading: Example

```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
```

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: Example

```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
```

CoGrammar

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)

CoGrammar

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)

Let’s get coding!

CoGrammar

Questions and Answers

CoGrammar

Let’s take a short break

CoGrammar

CoGrammar

SKILLS FOR LIFE
SKILLS BOOTCAMPS

Department for Education

Multidimensional Lists

April 2024

What are Lists?

- Picture organizing your bookshelf with various genres of books. In Python, lists act like shelves, helping you group similar items together. For instance, you can create a list of "fiction" books or "non-fiction" books.
- This makes it easy to manage and access your collection efficiently.

Lists Fundamentals

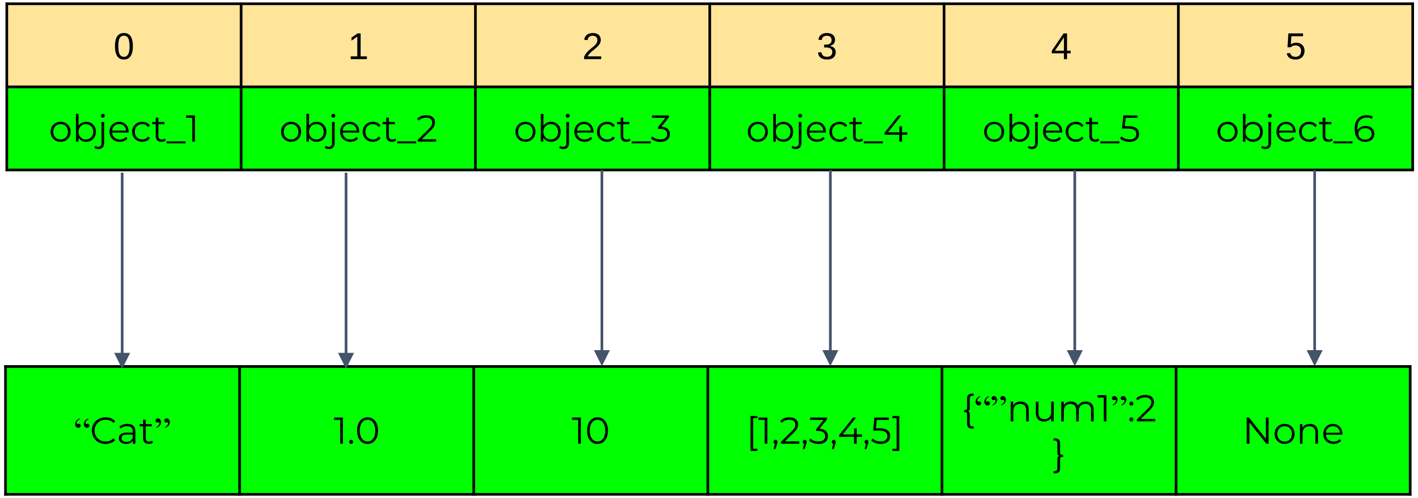
CoGrammar

Referential Lists

- As opposed to other programming languages like C++ or Java, **Python can receive various variable types in the same list.**
- Each cell in a list, stores the reference of each item inserted in it. Then inserting, retrieving and removing are done in quick time.

0	1	2	3	4	5
“Cat”	1.0	10	[1,2,3,4,5]	{“”num1”:2 }	None

Referential Lists



CoGrammar

Definitions

- **A container** is a construct used to group related values together and contains references to other objects instead of data.
- **A list** is a container created by surrounding a dynamically typed sequence of variables or literals with brackets [ ] or list().
- **An element** is a list item.
- **Index** in a list refers to the position of an element within the list. Usually starts from 0
- **Mutability** is the ability to modify a data structure at runtime. A list is a mutable data structure in Python.

CoGrammar

1D Lists

CoGrammar

1D Lists: Definition

A list is a container created by surrounding a dynamically typed sequence of variables or literals with brackets [ ] or list().

myList = [“cat”, 1.0, **10**, [1,2,3,4,5], {“”’num1”:2}, None]

myList[2] is 10

10 is at index 2

my_list = [ ] #or my_list = list()   #creates an empty list

CoGrammar

1D Lists: Operations

Adding an element in a list: append()

my_list = list() → Empty list

To add **3** to the list, then **5**

my_list**.append(3)** → 3 added to list

0
3

my_list**.append(5)** → 5 added to list

0	1
3	5

1D Lists: Operations

Removing an element in a list: pop()

my_list = list()

To add **3** to the list, then **5**

my_list**.append(3)**

my_list**.append(5)**

0	~~1~~
3	~~5~~

my_list.pop() # => returns **5**

0
3

1D Lists: Operations

Updating a cell in a list: update

my_list = list()

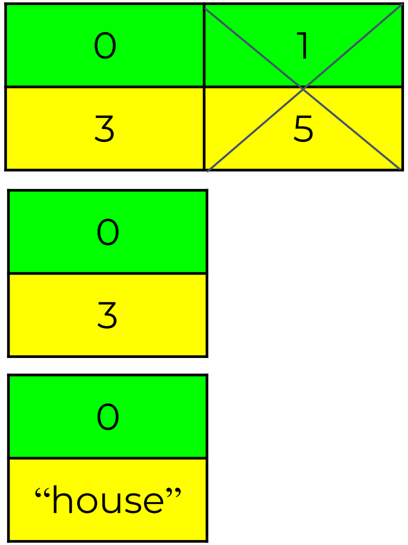
To add **3** to the list, then **5**

my_list**.append(3)**

my_list**.append(5)**

my_list.pop() # => returns **5**

My_list[0] = “house”



1D Lists: Operations

Extending the list: extend()

my_list[0] = “house”

your_list = [“Monday”, True]

0
“house”

Beware!

extend() is an **inplace** function

my_list.extend(your_list)

0	1	2
“house”	“Monday”	True

1D Lists: Operations

Extending the list: + (extend)

my_list[0] = “house”

your_list = [“Monday”, True]

0
“house”

Beware!

+ is not an **inplace** operation

new_list = my_list **+** your_list

0	1	2
“house”	“Monday”	True

2D Lists

CoGrammar

2D Lists: Definitions

Definitions

- A 2D list is an extension of a 1D List
- Each cell is an object referring to another list
- Two-dimensional lists, often referred to as 2D lists or matrices
- Nested Lists – List in a list

2D Lists: Operations

Access

>> new_list = [[1.0,"cat",3], [4,"fish",6], [7,"hen",9.0]]

- In 2D lists, we have 2 indices
- 1 index for the row
- 1 index for the column

	0	1	2
0	1.0	"cat"	3
1	4	"fish"	6
2	7	"hen"	9.0

To access "fish"
>> new_list[1][1]

3D Lists

CoGrammar

3D Lists: Operations

Access

```
>> new_list  = [
[['#', '#', '#'], ['#', '#', '#'], ['#', '#', '#']],
[['#', '#', '#'], ['#', '#', '#'], ['#', '#', '#']],
[['#', '#', '#'], ['#', '#', '#'], ['#', '#', '#']]
]
```

- In 3D lists, we have 3 indices
- 1 index for the row
- 1 index for the column
- 1 index for the third axis

matrix_item = [row_index][column_index][last_index]

Let’s get coding!

Questions and Answers

CoGrammar

Summary: Inheritance

- **Inheritance** is a fundamental concept in object-oriented programming (OOP) where a new class (subclass or derived class) is created from an existing class (superclass or base class).
- **Inheritance** facilitates code reuse and promotes the organization of code by allowing subclasses to inherit attributes and behaviours (methods) from their superclass.
- Understanding **inheritance** is crucial for effective object-oriented design and programming, as it forms the foundation for building modular, scalable, and maintainable software systems.

Summary: MD Lists

- **A list** is a container created by surrounding a dynamically typed list sequence of variables or literals with brackets [ ] or list().
- **Lists** operations include:
  - append()
  - pop()
  - extend()
- **Dimensionality** can be 1D, 2D, 3D and even deeper dimensions.

CoGrammar

Thank you for attending

SKILLS FOR LIFE
SKILLS BOOTCAMPS

Department for Education

CoGrammar