Let’s create a new class together and inherit from it. Below we have a base class called Clothing. Together, let’s create a second class, called Shirt, that inherits methods from the Clothing class. Fill in the blanks to make it work properly.

class Clothing:

  material = ""

  def \_\_init\_\_(self,name):

    self.name = name

  def checkmaterial(self):

    print("This {} is made of {}".format(self.name,self.material))

class Shirt(Clothing):

  material="Cotton"

polo = Shirt("Polo")

polo.checkmaterial()

RunReset

Here is your output:

This Polo is made of Cotton

Nice work! You used the existing Clothing class to make a

Shirt class that inherited from it!

Object Inheritance

In object-oriented programming, the concept of inheritance allows you to build relationships between objects, grouping together similar concepts and reducing code duplication. Let's create a custom Fruit class with color and flavor attributes:

>>> class Fruit:

...     def \_\_init\_\_(self, color, flavor):

...         self.color = color

...         self.flavor = flavor

...

We defined a Fruit class with a constructor for color and flavor attributes. Next, we'll define an Apple class along with a new Grape class, both of which we want to inherit properties and behaviors from the Fruit class:

>>> class Apple(Fruit):

...     pass

...

>>> class Grape(Fruit):

...     pass

In Python, we use parentheses in the class declaration to have the class inherit from the Fruit class. So in this example, we’re instructing our computer that both the Apple class and Grape class inherit from the Fruit class. This means that they both have the same constructor method which sets the color and flavor attributes. We can now create instances of our Apple and Grape classes:

6

>>> granny\_smith = Apple("green", "tart")

>>> carnelian = Grape("purple", "sweet")

>>> print(granny\_smith.flavor)

tart

>>> print(carnelian.color)

purple

Inheritance allows us to define attributes or methods that are shared by all types of fruit without having to define them in each fruit class individually. We can then also define specific attributes or methods that are only relevant for a specific type of fruit. Let's look at another example, this time with animals:

>>> class Animal:

...     sound = ""

...     def \_\_init\_\_(self, name):

...         self.name = name

...     def speak(self):

...         print("{sound} I'm {name}! {sound}".format(

...             name=self.name, sound=self.sound))

...

>>> class Piglet(Animal):

...     sound = "Oink!"

...

>>> class Cow(Animal):

...     sound = "Moooo"

...

We defined a parent class, Animal, with two animal types inheriting from that class: Piglet and Cow. The parent Animal class has an attribute to store the sound the animal makes, and the constructor class takes the name that will be assigned to the instance when it's created. There is also the speak method, which will print the name of the animal along with the sound it makes. We defined the Piglet and Cow classes, which inherit from the Animal class, and we set the sound attributes for each animal type. Now, we can create instances of our Piglet and Cow classes and have them speak:

>>> hamlet = Piglet("Hamlet")

>>> hamlet.speak()

Oink! I'm Hamlet! Oink!

...

>>> class Cow(Animal):

...     sound = "Moooo"

...

>>> milky = Cow("Milky White")

>>> milky.speak()

Moooo I'm Milky White! Moooo

We create instances of both the Piglet and Cow class, and set the names for our instances. Then we call the speak method of each instance, which results in the formatted string being printed; it includes the sound the animal type makes, along with the instance name we assigned.

Let’s expand a bit on our Clothing classes from the previous in-video question. Your mission: Finish the "Stock\_by\_Material" method and iterate over the amount of each item of a given material that is in stock. When you’re finished, the script should add up to 10 cotton Polo shirts.

class Clothing:

  stock={ 'name': [],'material' :[], 'amount':[]}

  def \_\_init\_\_(self,name):

    material = ""

    self.name = name

  def add\_item(self, name, material, amount):

    Clothing.stock['name'].append(self.name)

    Clothing.stock['material'].append(self.material)

    Clothing.stock['amount'].append(amount)

  def Stock\_by\_Material(self, material):

    count=0

    n=0

    for item in Clothing.stock['material']:

      if item == material:

        count += Clothing.stock['amount'][n]

        n+=1

    return count

class shirt(Clothing):

  material="Cotton"

class pants(Clothing):

  material="Cotton"

polo = shirt("Polo")

sweatpants = pants("Sweatpants")

polo.add\_item(polo.name, polo.material, 4)

sweatpants.add\_item(sweatpants.name, sweatpants.material, 6)

current\_stock = polo.Stock\_by\_Material("Cotton")

print(current\_stock)

RunReset

Here is your output:

10

Nice job! You successfully used composition to reuse the

Clothing.stock attribute and stock\_by\_material() function of

the Clothing class to take stock of the Cotton shirts!

Object Composition

You can have a situation where two different classes are related, but there is no inheritance going on. This is referred to as **composition** -- where one class makes use of code contained in another class. For example, imagine we have a **Package** class which represents a software package. It contains attributes about the software package, like name, version, and size. We also have a **Repository** class which represents all the packages available for installation. While there’s no inheritance relationship between the two classes, they are related. The Repository class will contain a dictionary or list of Packages that are contained in the repository. Let's take a look at an example Repository class definition:

class Repository:

...      def \_\_init\_\_(self):

...          self.packages = {}

...      def add\_package(self, package):

...          self.packages[package.name] = package

...      def total\_size(self):

...          result = 0

...          for package in self.packages.values():

...              result += package.size

...          return result

In the constructor method, we initialize the packages dictionary, which will contain the package objects available in this repository instance. We initialize the dictionary in the constructor to ensure that every instance of the Repository class has its own dictionary.

We then define the add\_package method, which takes a Package object as a parameter, and then adds it to our dictionary, using the package name attribute as the key.

Finally, we define a total\_size method which computes the total size of all packages contained in our repository. This method iterates through the values in our repository dictionary and adds together the size attributes from each package object contained in the dictionary, returning the total at the end. In this example, we’re making use of Package attributes within our Repository class. We’re also calling the values() method on our packages dictionary instance. Composition allows us to use objects as attributes, as well as access all their attributes and methods.

Let’s say we want to use the Keras deep learning module. Upon running the script, an error is printed stating the Keras module could not be found. What might we have missed?



We need to initialize the timedelta class



We need to define Keras functions



We need to define Keras attributes



We need to import the Keras module

**Correct**

You got it! We must use the import keyword to import the module before it can be used.

We need to import the Keras module

is selected.This is correct.

You got it! We must use the import keyword to import the module before it can be used.

Augmenting Python with Modules

Python modules are separate files that contain classes, functions, and other data that allow us to import and make use of these methods and classes in our own code. Python comes with a lot of modules out of the box. These modules are referred to as the Python Standard Library. You can make use of these modules by using the **import** keyword, followed by the module name. For example, we'll import the **random** module, and then call the **randint** function within this module:

>>> import random

>>> random.randint(1,10)

8

>>> random.randint(1,10)

7

>>> random.randint(1,10)

1

This function takes two integer parameters and returns a random integer between the values we pass it; in this case, 1 and 10. You might notice that calling functions in a module is very similar to calling methods in a class. We use dot notation here too, with a period between the module and function names.

Let's take a look at another module: **datetime**. This module is super helpful when working with dates and times.

>>> import datetime

>>> now = datetime.datetime.now()

>>> type(now)

<class 'datetime.datetime'>

>>> print(now)

2019-04-24 16:54:55.155199

First, we import the module. Next, we call the **now()** method which belongs to the **datetime** class contained within the **datetime** module. This method generates an instance of the datetime class for the current date and time. This instance has some methods which we can call:

>>> print(now)

2019-04-24 16:54:55.155199

>>> now.year

2019

>>> print(now + datetime.timedelta(days=28))

2019-05-22 16:54:55.155199

When we call the print function with an instance of the datetime class, we get the date and time printed in a specific format. This is because the datetime class has a **\_\_str\_\_** method defined which generates the formatted string we see here. We can also directly call attributes and methods of the class, as with **now.year** which returns the year attribute of the instance.

Lastly, we can access other classes contained in the datetime module, like the **timedelta** class. In this example, we’re creating an instance of the timedelta class with the parameter of 28 days. We’re then adding this object to our instance of the datetime class from earlier and printing the result. This has the effect of adding 28 days to our original datetime object.

Supplemental Reading for Code Reuse (Optional)

Supplemental Reading for Code Reuse

The official Python documentation lists all the modules included in the standard library. It even has a turtle in it...

[Pypi](https://pypi.org/) is the Python repository and index of an impressive number of modules developed by Python programmers around the world.