# Learn Python: Classes

About 2 hours, 25 minutes

Let's learn about Python Classes!

Python comes with many built-in data types for our needs, but sometimes they need a little extra oomph for whatever program we're designing! Learn how to create your own data types in this lesson.

# Types

Python equips us with many different ways to store data. A float is a different kind of number from an int, and we store different data in a list than we do in a dict. These are known as different types. We can check the type of a Python variable using the type() function.

a\_string = "Cool String"

an\_int = 12

print(type(a\_string))

# prints "<class 'str'>"

print(type(an\_int))

# prints "<class 'int'>"

Above, we defined two variables, and checked the type of these two variables. A variable's type determines what you can do with it and how you can use it. You can't .get() something from an integer, just as you can't add two dictionaries together using +. This is because those operations are defined at the type level.

**1.**

Call type() on the integer 5 and print the results.

**2.**

Define a dictionary my\_dict.

**3.**

Print out the type() of my\_dict.

**4.**

Define a list called my\_list.

**5.**

Print out the type() of my\_list.

**Solution**

type(5)

print(type(5))

my\_dict = {}

print(type(my\_dict))

my\_list = []

print(type(my\_list))

# Class

A class is a template for a data type. It describes the kinds of information that class will hold and how a programmer will interact with that data. Define a class using the class keyword. [PEP 8 Style Guide for Python Code](https://www.python.org/dev/peps/pep-0008/#class-names) recommends capitalizing the names of classes to make them easier to identify.

class CoolClass:

pass

In the above example we created a class and named it CoolClass. We used the pass keyword in Python to indicate that the body of the class was intentionally left blank so we don't cause an IndentationError. We’ll learn about all the things we can put in the body of a class in the next few exercises.

Instructions

**1.**

Define an empty class called Facade. We'll chip away at it soon!

**Solution:**

class Facade:

pass

# Instantiation

A class doesn't accomplish anything simply by being defined. A class must be instantiated. In other words, we must create an instance of the class, in order to breathe life into the schematic.

Instantiating a class looks a lot like calling a function. We would be able to create an instance of our defined CoolClass as follows:

cool\_instance = CoolClass()

Above, we created an object by adding parentheses to the name of the class. We then assigned that new instance to the variable cool\_instance for safe-keeping.

Instructions

**1.**

In **script.py** we see our Facade class from last exercise. Make a Facade instance and save it to the variable facade\_1.

**Solution:**

class Facade:

pass

facade\_1 = Facade()

# Object-Oriented Programming

A class instance is also called an object. The pattern of defining classes and creating objects to represent the responsibilities of a program is known as Object Oriented Programming or OOP.

Instantiation takes a class and turns it into an object, the type() function does the opposite of that. When called with an object, it returns the class that the object is an instance of.

print(type(cool\_instance))

# prints "<class '\_\_main\_\_.CoolClass'>"

We then print out the type() of cool\_instance and it shows us that this object is of type \_\_main\_\_.CoolClass.

In Python \_\_main\_\_ means "this current file that we're running" and so one could read the output from type() to mean "the class CoolClass that was defined here, in the script you're currently running."

Instructions

**1.**

In **script.py** we see facade\_1 from last exercise. Try calling type() on facade\_1 and saving it to the variable facade\_1\_type.

**2.**

Print out facade\_1\_type.

**Solution:**

class Facade:

pass

facade\_1 = Facade()

facade\_1\_type = type(facade\_1)

print(facade\_1\_type)

# Class Variables

When we want the same data to be available to every instance of a class we use a class variable. A class variable is a variable that's the same for every instance of the class.

You can define a class variable by including it in the indented part of your class definition, and you can access all of an object's class variables with object.variable syntax.

class Musician:

title = "Rockstar"

drummer = Musician()

print(drummer.title)

# prints "Rockstar"

Above we defined the class Musician, then instantiated drummer to be an object of type Musician. We then printed out the drummer's .title attribute, which is a class variable that we defined as the str

ing "Rockstar".

If we defined another musician, like guitarist = Musician() they would have the same .title attribute.

Instructions

**1.**

You are digitizing grades for Jan van Eyck High School and Conservatory. At Jan van High, as the students call it, 65 is the minimum passing grade.

Create a Grade class with a class attribute minimum\_passing equal to 65.

**Hint:**

Class variables are defined in the body of a class definition.

class CoolClass:

class\_variable = True

**Solution:**

class Grade:

minimum\_passing = 65

# Methods

Methods are functions that are defined as part of a class. The first argument in a method is always the object that is calling the method. Convention recommends that we name this first argument self. Methods always have at least this one argument.

We define methods similarly to functions, except that they are indented to be part of the class.

class Dog():

dog\_time\_dilation = 7

def time\_explanation(self):

print("Dogs experience {} years for every 1 human year.".format(self.dog\_time\_dilation))

pipi\_pitbull = Dog()

pipi\_pitbull.time\_explanation()

# Prints "Dogs experience 7 years for every 1 human year."

Above we created a Dog class with a time\_explanation method that takes one argument, self, which refers to the object calling the function. We created a Dog named pipi\_pitbull and called the .time\_explanation() method on our new object for Pipi.

Notice we didn't pass any arguments when we called .time\_explanation(), but were able to refer to self in the function body. When you call a method it automatically passes the object calling the method as the first argument.

Instructions

**1.**

At Jan van High, the students are constantly calling the school rules into question. Create a Rules class so that we can explain the rules.

Remember to create a class you need to use the class keyword.

**2.**

Give Rules a method washing\_brushes that returns the string .

"Point bristles towards the basin while washing your brushes."

Be sure to add self as a parameter when defining a method.

class CoolClass:

def cool\_method(self):

print("Cool!")

And use return to return the given string in your method.

**Solution:**

class Rules:

def washing\_brushes(self):

return "Point bristles towards the basin while washing your brushes."

# Methods with Arguments

Methods can also take more arguments than just self:

class DistanceConverter:

kms\_in\_a\_mile = 1.609

def how\_many\_kms(self, miles):

return miles \* self.kms\_in\_a\_mile

converter = DistanceConverter()

kms\_in\_5\_miles = converter.how\_many\_kms(5)

print(kms\_in\_5\_miles)

# prints "8.045"

Above we defined a DistanceConverter class, instantiated it, and used it to convert 5 miles into kilometers. Notice again that even though how\_many\_kms takes two arguments in its definition, we only pass miles, because self is implicitly passed (and refers to the object converter).

Instructions

**1.**

It's March 14th (known in some places as **Pi day**) at Jan van High, and you're feeling awfully festive. You decide to create a program that calculates the area of a circle.

Create a Circle class with class variable pi. Set pi to the approximation 3.14.

**2.**

Give Circle an area method that takes two parameters: self and radius.

Return the area as given by this formula:

area = pi \* radius \*\* 2

Since pi is a class variable, you can access it as an attribute of the class.

area = Circle.pi \* radius \*\* 2

**3.**

Create an instance of Circle. Save it into the variable circle.

**4.**

You go to measure several circles you happen to find around.

* A medium pizza that is 12 inches across.
* Your teaching table which is 36 inches across.
* The Round Room auditorium, which is 11,460 inches across.

You save the areas of these three things into pizza\_area, teaching\_table\_area, and round\_room\_area.

Remember that the radius of a circle is half the diameter. We gave three diameters here, so halve them before you calculate the given circle's area.

Given a certain diameter, calculate the area using

circle.area(diameter / 2)

**Solution:**

class Circle:

pi = 3.14

def area(self, radius):

return Circle.pi \* radius \*\* 2

circle = Circle()

pizza\_area = circle.area(12 / 2)

teaching\_table\_area = circle.area(36 / 2)

round\_room\_area = circle.area(11460 / 2)

# Constructors

There are several methods that we can define in a Python class that have special behavior. These methods are sometimes called "magic", because they behave differently from regular methods. Another popular term is dunder methods, so-named because they have two underscores (double-underscore abbreviated to "dunder") on either side of them.

The first dunder method we're going to use is the \_\_init\_\_ method (note the two underscores before and after the word "init"). This method is used to initialize a newly created object. It is called every time the class is instantiated.

Methods that are used to prepare an object being instantiated are called constructors. The word "constructor" is used to describe similar features in other object-oriented programming languages but programmers who refer to a constructor in Python are usually talking about the \_\_init\_\_ method.

class Shouter:

def \_\_init\_\_(self):

print("HELLO?!")

shout1 = Shouter()

# prints "HELLO?!"

shout2 = Shouter()

# prints "HELLO?!"

Above we created a class called Shouter and every time we create an instance of Shouter the program prints out a shout. Don't worry, this doesn't hurt the computer at all.

Pay careful attention to the instantiation syntax we use. Shouter() looks a lot like a function call, doesn't it? If it's a function, can we pass parameters to it? We absolutely can, and those parameters will be received by the \_\_init\_\_ method.

class Shouter:

def \_\_init\_\_(self, phrase):

# make sure phrase is a string

if type(phrase) == str:

# then shout it out

print(phrase.upper())

shout1 = Shouter("shout")

# prints "SHOUT"

shout2 = Shouter("shout")

# prints "SHOUT"

shout3 = Shouter("let it all out")

# prints "LET IT ALL OUT"

Above we've updated our Shouter class to take the additional parameter phrase. When we created each of our objects we passed an argument to the constructor. The constructor takes the argument phrase and, if it's a string, prints out the all-caps version of phrase.

**1.**

Add a constructor to our Circle class.

Since we seem more frequently to know the diameter of a circle, it should take the argument diameter.

It doesn't need to do anything yet, just write pass in the body of the constructor.

**2.**

Now have the constructor print out the message "New circle with diameter: {diameter}" when a new circle is created.

Create a circle teaching\_table with diameter 36.

**Solution:**

class Circle:

pi = 3.14

# Add constructor here:

def \_\_init\_\_(self, diameter):

print("New circle with diameter: {diameter}".format(diameter=diameter))

teaching\_table = Circle(36)

# Instance Variables

We've learned so far that a class is a schematic for a data type and an object is an instance of a class, but why is there such a strong need to differentiate the two if each object can only have the methods and class variables the class has? This is because each instance of a class can hold different kinds of data.

The data held by an object is referred to as an instance variable. Instance variables aren't shared by all instances of a class — they are variables that are specific to the object they are attached to.

Let's say that we have the following class definition:

class FakeDict:

pass

We can instantiate two different objects from this class, fake\_dict1 and fake\_dict2, and assign instance variables to these objects using the same attribute notation that was used for accessing class variables.

fake\_dict1 = FakeDict()

fake\_dict2 = FakeDict()

fake\_dict1.fake\_key = "This works!"

fake\_dict2.fake\_key = "This too!"

# Let's join the two strings together!

working\_string = "{} {}".format(fake\_dict1.fake\_key, fake\_dict2.fake\_key)

print(working\_string)

# prints "This works! This too!"

**1.**

In **script.py** we have defined a Store class. Create two objects from this store class, named alternative\_rocks and isabelles\_ices.

**2.**

Give them both instance attributes called store\_name. Set alternative\_rocks's store\_name to "Alternative Rocks". Set isabelles\_ices's store\_name to "Isabelle's Ices".

**Solution:**

class Store:

pass

alternative\_rocks = Store()

isabelles\_ices = Store()

alternative\_rocks.store\_name = "Alternative Rocks"

isabelles\_ices.store\_name = "Isabelle's Ices"

# Attribute Functions

Instance variables and class variables are both accessed similarly in Python. This is no mistake, they are both considered attributes of an object. If we attempt to access an attribute that is neither a class variable nor an instance variable of the object Python will throw an AttributeError.

class NoCustomAttributes:

pass

attributeless = NoCustomAttributes()

try:

attributeless.fake\_attribute

except AttributeError:

print("This text gets printed!")

# prints "This text gets printed!"

What if we aren't sure if an object has an attribute or not? getattr() is a Python function that works a lot like the usual dot-syntax (i.e., object\_name.attribute\_name) but we can supply a third argument that will be the default if the object does not have the given attribute. What if we only really care whether the attribute exists? hasattr() will return True if an object has a given attribute and False otherwise.

Calling those functions looks like this:

hasattr(attributeless, "fake\_attribute")

# returns False

getattr(attributeless, "other\_fake\_attribute", 800)

# returns 800, the default value

Above we checked if the attributeless object has the attribute fake\_attribute. Since it does not, hasattr() returned False. After that, we used getattr to attempt to retrieve other\_fake\_attribute. Since other\_fake\_attribute isn't a real attribute on attributeless, our call to getattr() returned the supplied default value 800, instead of throwing an AttributeError.

Instructions

**1.**

In **script.py** we have a list of different data types, some strings, some lists, and some dictionaries, all saved in the variable how\_many\_s.

For every element in the list, check if the element has the attribute count. If so, count the number of times the string "s" appears in the element. Print this number.

Try something like this:

for element in list\_of\_things:

if hasattr(element, "special\_method"):

print(element.special\_method(argument))

**Solution:**

how\_many\_s = [{'s': False}, "sassafrass", 18, ["a", "c", "s", "d", "s"]]

for element in how\_many\_s:

if hasattr(element, "count"):

print(element.count("s"))

# Self

Since we can already use dictionaries to store key-value pairs, using objects for that purpose is not really useful. Instance variables are more powerful when you can guarantee a rigidity to the data the object is holding.

This convenience is most apparent when the constructor creates the instance variables, using the arguments passed in to it. If we were creating a search engine, and we wanted to create classes for each separate entry we could return. We'd do that like this:

class SearchEngineEntry:

def \_\_init\_\_(self, url):

self.url = url

codecademy = SearchEngineEntry("www.codecademy.com")

wikipedia = SearchEngineEntry("www.wikipedia.org")

print(codecademy.url)

# prints "www.codecademy.com"

print(wikipedia.url)

# prints "www.wikipedia.org"

Since the self keyword refers to the object and not the class being called, we can define a secure method on the SearchEngineEntry class that returns the secure link to an entry.

class SearchEngineEntry:

secure\_prefix = "https://"

def \_\_init\_\_(self, url):

self.url = url

def secure(self):

return "{prefix}{site}".format(prefix=self.secure\_prefix, site=self.url)

codecademy = SearchEngineEntry("www.codecademy.com")

print(codecademy.secure())

# prints "https://www.codecademy.com"

print(wikipedia.secure())

# prints "https://www.wikipedia.org"

Above we define our secure() method to take just the one required argument, self. We access both the class variable self.secure\_prefix and the instance variable self.url to return a secure URL.

This is the strength of writing object-oriented programs. We can write our classes to structure the data that we need and write methods that will interact with that data in a meaningful way.

Instructions

**1.**

In **script.py** you'll find our familiar friend, the Circle class.

Even though we usually know the diameter beforehand, what we need for most calculations is the radius.

In Circle's constructor set the instance variable self.radius to equal half the diameter that gets passed in.

**2.**

Define three Circles with three different diameters.

* A medium pizza, medium\_pizza, that is 12 inches across.
* Your teaching table, teaching\_table, which is 36 inches across.
* The Round Room auditorium, round\_room, which is 11,460 inches across.

**3.**

Define a new method circumference for your circle object that takes only one argument, self, and returns the circumference of a circle with the given radius by this formula:

circumference = 2 \* pi \* radius

Remember to use self to refer to class and instance variables in a method.

2 \* self.pi \* self.radius

**4.**

Print out the circumferences of medium\_pizza, teaching\_table, and round\_room.

**Solution:**

class Circle:

pi = 3.14

def \_\_init\_\_(self, diameter):

print("Creating circle with diameter {d}".format(d=diameter))

# Add assignment for self.radius here:

self.radius = diameter / 2

def circumference(self):

return 2 \* self.pi \* self.radius

medium\_pizza = Circle(12)

teaching\_table = Circle(36)

round\_room = Circle(11460)

print(medium\_pizza.circumference())

print(teaching\_table.circumference())

print(round\_room.circumference())

**Everything is an Object**

Attributes can be added to user-defined objects after instantiation, so it's possible for an object to have some attributes that are not explicitly defined in an object's constructor. We can use the dir() function to investigate an object's attributes at runtime. dir() is short for *directory* and offers an organized presentation of object attributes.

class FakeDict:

pass

fake\_dict = FakeDict()

fake\_dict.attribute = "Cool"

dir(fake\_dict)

# Prints ['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', 'attribute']

That's certainly a lot more attributes than we defined! Python automatically adds a number of attributes to all objects that get created. These internal attributes are usually indicated by double-underscores. But sure enough, attribute is in that list.

Do you remember being able to use type() on Python's native data types? This is because they are also objects in Python. Their classes are int, float, str, list, and dict. These Python classes have special syntax for their instantiation, 1, 1.0, "hello", [], and {} specifically. But these instances are still full-blown objects to Python.

fun\_list = [10, "string", {'abc': True}]

type(fun\_list)

# Prints <class 'list'>

dir(fun\_list)

# Prints ['\_\_add\_\_', '\_\_class\_\_', [...], 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']

Above we define a new list. We check it's type and see that's an instantiation of class list. We use dir() to explore its attributes, and it gives us a large number of internal Python dunder attributes, but, afterward, we get the usual list methods.

Instructions

**1.**

Call dir() on the number 5. Print out the results.

Print the results from calling dir(5).

**2.**

Define a function called this\_function\_is\_an\_object. It can take any parameters and return anything you'd like.

**3.**

Print out the result of calling dir() on this\_function\_is\_an\_object.

Functions are objects too!

**Solution:**

print(dir(5))

def this\_function\_is\_an\_object(num):

return "Cheese is {} times better than everything else".format(num)

print(dir(this\_function\_is\_an\_object))

# String Representation

One of the first things we learn as programmers is how to print out information that we need for debugging. Unfortunately, when we print out an object we get a default representation that seems fairly useless.

class Employee():

def \_\_init\_\_(self, name):

self.name = name

argus = Employee("Argus Filch")

print(argus)

# prints "<\_\_main\_\_.Employee object at 0x104e88390>"

This default string representation gives us some information, like where the class is defined and our computer's memory address where this object is stored, but is usually not useful information to have when we are trying to debug our code.

We learned about the dunder method \_\_init\_\_. Now, we will learn another dunder method called \_\_repr\_\_. This is a method we can use to tell Python what we want the string representation of the class to be. \_\_repr\_\_ can only have one parameter, self, and must return a string.

In our Employee class above, we have an instance variable called name that should be unique enough to be useful when we're printing out an instance of the Employee class.

class Employee():

def \_\_init\_\_(self, name):

self.name = name

def \_\_repr\_\_(self):

return self.name

argus = Employee("Argus Filch")

print(argus)

# prints "Argus Filch"

We implemented the \_\_repr\_\_ method and had it return the .name attribute of the object. When we printed the object out it simply printed the .name of the object! Cool!

Instructions

**1.**

Add a \_\_repr\_\_ method to the Circle class that returns

Circle with radius {radius}

**2.**

Print out medium\_pizza, teaching\_table, and round\_room.

**Solution:**

class Circle:

pi = 3.14

def \_\_init\_\_(self, diameter):

self.radius = diameter / 2

def area(self):

return self.pi \* self.radius \*\* 2

def circumference(self):

return self.pi \* 2 \* self.radius

def \_\_repr\_\_(self):

return "Circle with radius {radius}".format(radius=self.radius)

medium\_pizza = Circle(12)

teaching\_table = Circle(36)

round\_room = Circle(11460)

print(medium\_pizza)

print(teaching\_table)

print(round\_room)

**Review**

So far we've covered what a data type actually is in Python. We explored what the functionality of Python's built-in types (also referred to as *primitives*) are. We learned how to create our own data types using the class keyword.

We explored the relationship between a class and an object — we create objects when we instantiate a class, we find the class when we check the type() of an object. We learned the difference between class variables (the same for all objects of a class) and instance variables (unique for each object).

We learned about how to define an object's functionality with methods. We created multiple objects from the same class, all with similar functionality, but with different internal data. They all had the same methods, but produced different output because they were different instances.

Take a moment to congratulate yourself, object-oriented programming is a complicated concept.

Instructions

**1.**

Define a class Student this will be our data model at *Jan van Eyck High School and Conservatory*.

**2.**

Add a constructor for Student. Have the constructor take in two parameters: a name and a year. Save those two as attributes .name and .year.

Create a constructor by defining the method \_\_init\_\_. Make sure it takes three arguments: self, name, and year.

class CoolClass:

def \_\_init\_\_(self, param1):

self.attr1 = param1

**3.**

Create three instances of the Student class:

* Roger van der Weyden, year 10
* Sandro Botticelli, year 12
* Pieter Bruegel the Elder, year 8

Save them into the variables roger, sandro, and pieter.

Create an instance by passing arguments to the class constructor.

cool\_object = CoolClass(cool\_arg1, cool\_arg2)

**4.**

Create a Grade class, with minimum\_passing as an attribute set to 65.

**5.**

Give Grade a constructor. Take in a parameter score and assign it to self.score.

**6.**

In the body of the constructor for Student, declare self.grades as an empty list.

**7.**

Add an .add\_grade() method to Student that takes a parameter, grade.

.add\_grade() should verify that grade is of type Grade and if so, add it to the Student's .grades.

If grade isn't an instance of Grade then .add\_grade() should do nothing.

Remember you can check an object's type using the type() function.

if type(cool\_object) == CoolClass:

do\_something

**8.**

Create a new Grade with a score of 100 and add it to pieter's .grades attribute using .add\_grade().

**9.**

Great job! You've created two classes and defined their interactions. This is object-oriented programming! From here you could:

* Write a Grade method .is\_passing() that returns whether a Grade has a passing .score.
* Write a Student method get\_average() that returns the student's average score.
* Add an instance variable to Student that is a dictionary called .attendance, with dates as keys and booleans as values that indicate whether the student attended school that day.
* Write your own classes to do whatever logic you want!

**Solution:**

class Student:

def \_\_init\_\_(self, name, year):

self.name = name

self.year = year

self.grades = []

def add\_grade(self, grade):

if type(grade) is Grade:

self.grades.append(grade)

class Grade:

minimum\_passing = 65

def \_\_init\_\_(self, score):

self.score = score

roger = Student("Roger van der Weyden", 10)

sandro = Student("Sandro Botticelli", 12)

pieter = Student("Pieter Bruegel the Elder", 8)

pieter.add\_grade(Grade(100))

# Learn Python: Inheritance and Polymorphism

About 2 hours

Welcome to Learn Python: Inheritance and Polymorphism!

In this lesson we learn about some more advanced Object Oriented concepts. We talk about building your classes so that they have reusable interfaces and we explore how to make our classes behave more like Python's native data types.

# Inheritance

Classes are designed to allow for more code reuse, but what if we need a class that looks a lot like a class we already have? If the bulk of a class's definition is useful, but we have a new use case that is distinct from how the original class was used, we can inherit from the original class. Think of inheritance as a remix — it sounds a lot like the original, but there's something... different about it.

class User:

is\_admin = False

def \_\_init\_\_(self, username)

self.username = username

class Admin(User):

is\_admin = True

Above we defined User as our base class. We want to create a new class that inherits from it, so we created the subclass Admin. In the above example, Admin has the same constructor as User. Only the class variable is\_admin is set differently between the two.

Sometimes a base class is called a parent class. In these terms, the class inheriting from it, the subclass, is also referred to as a child class.

Instructions

**1.**

In **script.py** we've already defined the class Bin. Create a subclass of Bin called RecyclingBin.

Use this syntax to subclass an existing class:

class Subclass(Superclass):

pass

**Solution:**

class Bin:

pass

class RecyclingBin(Bin):

pass

# Exceptions

There's one very important family of class definitions built in to the Python language. An Exception is a class that inherits from Python's Exception class.

We can validate this ourselves using the issubclass() function. issubclass() is a Python built-in function that takes two parameters. issubclass() returns True if the first argument is a subclass of the second argument. It returns False if the first class is not a subclass of the second. issubclass() raises a TypeError if either argument passed in is not a class.

issubclass(ZeroDivisionError, Exception)

# Returns True

Above, we checked whether ZeroDivisionError, the exception raised when attempting division by zero, is a subclass of Exception. It is, so issubclass returns True.

Why is it beneficial for exceptions to inherit from one another? Let's consider an example where we create our own exceptions. What if we were creating software that tracks our kitchen appliances? We would be able to design a suite of exceptions for that need:

class KitchenException(Exception):

"""

Exception that gets thrown when a kitchen appliance isn't working

"""

class MicrowaveException(KitchenException):

"""

Exception for when the microwave stops working

"""

class RefrigeratorException(KitchenException):

"""

Exception for when the refrigerator stops working

"""

In this code, we define three exceptions. First, we define a KitchenException that acts as the parent to our other, specific kitchen appliance exceptions. KitchenException subclasses Exception, so it behaves in the same way that regular Exceptions do. Afterward we define MicrowaveException and RefrigeratorException as subclasses.

Since our exceptions are subclassed in this way, we can catch any of KitchenException's subclasses by catching KitchenException. For example:

def get\_food\_from\_fridge():

if refrigerator.cooling == False:

raise RefrigeratorException

else:

return food

def heat\_food(food):

if microwave.working == False:

raise MicrowaveException

else:

microwave.cook(food)

return food

try:

food = get\_food\_from\_fridge()

food = heat\_food(food)

except KitchenException:

food = order\_takeout()

In the above example, we attempt to retrieve food from the fridge and heat it in the microwave. If either RefrigeratorException or MicrowaveException is raised, we opt to order takeout instead. We catch both RefrigeratorException and MicrowaveException in our try/except block because both are subclasses of KitchenException.

Explore Python's exception hierarchy in the [Python documentation](https://docs.python.org/3/library/exceptions.html#exception-hierarchy)!

Instructions

**1.**

In **script.py** we've defined a CandleShop class for our new candle shop that we've named Here's a Hot Tip: Buy Drip Candles. We want to define our own exceptions for when we run out of candles to sell.

Define your own exception called OutOfStock that inherits from the Exception class.

Remember subclassing a class involves using parentheses syntax

class Subclass(Superclass):

pass

Use the Python built-in Exception as your superclass and you'll define an exception!

**2.**

Have CandleShop raise your OutOfStock exception when CandleShop.buy() tries to buy a candle that's out of stock.

We reject code that raise exceptions, so don't raise OutOfStock explicitly outside of the .buy() method in the CandleShop class.

**Solution:**

# Define your exception up here:

class OutOfStock(Exception):

pass

# Update the class below to raise OutOfStock

class CandleShop:

name = "Here's a Hot Tip: Buy Drip Candles"

def \_\_init\_\_(self, stock):

self.stock = stock

def buy(self, color):

if self.stock[color] < 1:

raise OutOfStock

self.stock[color] = self.stock[color] - 1

candle\_shop = CandleShop({'blue': 6, 'red': 2, 'green': 0})

candle\_shop.buy('blue')

# This should raise OutOfStock:

# candle\_shop.buy('green')

# Overriding Methods

Inheritance is a useful way of creating objects with different class variables, but is that all it's good for? What if one of the methods needs to be implemented differently? In Python, all we have to do to override a method definition is to offer a new definition for the method in our subclass.

An overridden method is one that has a different definition from its parent class. What if User class didn't have an is\_admin flag but just checked if it had permission for something based on a permissions dictionary? It could look like this:

class User:

def \_\_init\_\_(self, username, permissions):

self.username = username

self.permissions = permissions

def has\_permission\_for(self, key):

if self.permissions.get(key):

return True

else:

return False

Above we defined a class User which takes a permissions parameter in its constructor. Let's assume permissions is a dict. User has a method .has\_permission\_for() implemented, where it checks to see if a given key is in its permissions dictionary. We could then define our Admin user like this:

class Admin(User):

def has\_permission\_for(self, key):

return True

Here we define an Admin class that subclasses User. It has all methods, attributes, and functionality that User has. However, if you call has\_permission\_for on an instance of Admin, it won't check its permissions dictionary. Since this User is also an Admin, we just say they have permission to see everything!

Instructions

**1.**

In **script.py**, we've defined two classes, Message and User. Create an Admin class that subclasses the User class.

**2.**

Override User's .edit\_message() method in Admin so that an Admin can edit any messages.

**Solution:**

class Message:

def \_\_init\_\_(self, sender, recipient, text):

self.sender = sender

self.recipient = recipient

self.text = text

class User:

def \_\_init\_\_(self, username):

self.username = username

def edit\_message(self, message, new\_text):

if message.sender == self.username:

message.text = new\_text

class Admin(User):

def edit\_message(self, message, new\_text):

message.text = new\_text

Learn Python: Inheritance and Polymorphism

# Super()

Overriding methods is really useful in some cases but sometimes we want to add some extra logic to the existing method. In order to do that we need a way to call the method from the parent class. Python gives us a way to do that using super().

super() gives us a proxy object. With this proxy object, we can invoke the method of an object's parent class (also called its superclass). We call the required function as a method on super():

class Sink:

def \_\_init\_\_(self, basin, nozzle):

self.basin = basin

self.nozzle = nozzle

class KitchenSink(Sink):

def \_\_init\_\_(self, basin, nozzle, trash\_compactor=None):

super().\_\_init\_\_(self, basin, nozzle)

if trash\_compactor:

self.trash\_compactor = trash\_compactor

Above we defined two classes. First, we defined a Sink class with a constructor that assigns a rinse basin and a sink nozzle to a Sink instance. Afterwards, we defined a KitchenSink class that inherits from Sink. KitchenSink's constructor takes an additional parameter, a trash\_compactor. KitchenSink then calls the constructor for Sink with the basin and nozzle it received using the super() function, with this line of code:

super().\_\_init\_\_(self, basin, nozzle)

This line says "call the constructor (the function called \_\_init\_\_) of the class that is this class's parent class. In the example given, KitchenSink's constructor calls the constructor for Sink. In this way, we can override a parent class's method to add some new functionality (like adding a trash\_compactor to a sink), while still retaining the behavior of the original constructor (like setting the basin and nozzle as instance variables).

Instructions

**1.**

You're invited to a potluck this week and decide to make your own special version of Potato Salad!

In **script.py** you'll find a class called PotatoSalad, make a subclass of PotatoSalad called SpecialPotatoSalad.

**2.**

Your special potato salad recipe is pretty similar to a regular potato salad, so let's start with making that.

Create a new constructor for SpecialPotatoSalad that just calls the parent constructor for PotatoSalad. Make sure that SpecialPotatoSalad's constructor takes the same arguments as PotatoSalad.

Create an \_\_init\_\_ method for SpecialPotatoSalad with the same arguments as PotatoSalad.

In the \_\_init\_\_ method, call the parent class's method with this syntax:

class SubClass(ParentClass):

def inherited\_method(self, arg1, arg2):

super().inherited\_method(arg1, arg2)

**3.**

The difference with your special potato salad is... you add raisins to it! You can't remember when you started doing this, but Dolores always hoots about it at her potlucks and if that isn't the nicest thing. You always use the full package of raisins for every potato salad you make, and each package has 40 raisins in it.

In your constructor for SpecialPotatoSalad, after making regular potato salad, set self.raisins = 40.

**Solution:**

class PotatoSalad:

def \_\_init\_\_(self, potatoes, celery, onions):

self.potatoes = potatoes

self.celery = celery

self.onions = onions

class SpecialPotatoSalad(PotatoSalad):

def \_\_init\_\_(self, potatoes, celery, onions):

super().\_\_init\_\_(potatoes, celery, onions)

self.raisins = 40

**Interfaces**

You may be wondering at this point why we would even want to have two different classes with two differently implemented methods to use the same method name. This style is especially useful when we have an object for which it might not matter which class the object is an instance of. Instead, we're interested in whether that object can perform a given task.

If we have the following code:

class Chess:

def \_\_init\_\_(self):

self.board = setup\_board()

self.pieces = add\_chess\_pieces()

def play(self):

print("Playing chess!")

class Checkers:

def \_\_init\_\_(self):

self.board = setup\_board()

self.pieces = add\_checkers\_pieces()

def play(self):

print("Playing checkers!")

In the code above we define two classes, Chess and Checkers. In Chess we define a constructor that sets up the board and pieces, and a .play() method. Checkers also defines a .play() method. If we have a play\_game() function that takes an instance of Chess or Checkers, it could call the .play() method without having to check which class the object is an instance of.

def play\_game(chess\_or\_checkers):

chess\_or\_checkers.play()

chess\_game = Chess()

checkers\_game = Checkers()

chess\_game\_2 = Chess()

for game in [chess\_game, checkers\_game, chess\_game\_2]:

play\_game(game)

"""

Prints out the following:

Playing chess!

Playing checkers!

Playing chess!

In this code, we defined a play\_game function that could take either a Chess object or a Checkers object. We instantiate a few objects and then call play\_game on each.

When two classes have the same method names and attributes, we say they implement the same *interface*. An interface in Python usually refers to the names of the methods and the arguments they take. Other programming languages have more rigid definitions of what an interface is, but it usually hinges on the fact that different objects from different classes can perform the same operation (even if it is implemented differently for each class).

Instructions

**1.**

In **script.py** we've defined an InsurancePolicy class. Create a subclass of InsurancePolicy called VehicleInsurance.

**2.**

Create a different subclass of InsurancePolicy called HomeInsurance.

**3.**

Give VehicleInsurance a .get\_rate() method that takes self as a parameter. Return .001 multiplied by the price of the vehicle.

**4.**

Give HomeInsurance a .get\_rate() method that takes self as a parameter. Return .00005 multiplied by the price of the home.

**Solution:**

class InsurancePolicy:

def \_\_init\_\_(self, price\_of\_item):

self.price\_of\_insured\_item = price\_of\_item

class VehicleInsurance(InsurancePolicy):

def get\_rate(self):

return self.price\_of\_insured\_item \* .001

class HomeInsurance(InsurancePolicy):

def get\_rate(self):

return self.price\_of\_insured\_item \* .00005

# Polymorphism

All this talk of interfaces demonstrates flexibility in programming. Flexibility in programming is a broad philosophy, but what's worth remembering is that we want to implement forms that are familiar in our programs so that usage is expected. For example, let's think of the + operator. It's easy to think of it as a single function that "adds" whatever is on the left with whatever is on the right, but it does many different things in different contexts:

# For an int and an int, + returns an int

2 + 4 == 6

# For a float and a float, + returns a float

3.1 + 2.1 == 5.2

# For a string and a string, + returns a string

"Is this " + "addition?" == "Is this addition?"

# For a list and a list, + returns a list

[1, 2] + [3, 4] == [1, 2, 3, 4]

Look at all the different things that + does! The hope is that all of these things are, for the arguments given to them, the intuitive result of adding them together. Polymorphism is the term used to describe the same syntax (like the + operator here, but it could be a method name) doing different actions depending on the type of data.

Polymorphism is an abstract concept that covers a lot of ground, but defining class hierarchies that all implement the same interface is a way of introducing polymorphism to our code.

Instructions

**1.**

In **script.py** a few different types of data are provided. Call len() on each of them.

Is the same operation happening for each? How is it different? How is it similar? Does using len() to refer to these different operations make sense?

**Solution:**

a\_list = [1, 18, 32, 12]

a\_dict = {'value': True}

a\_string = "Polymorphism is cool!"

len(a\_list)

len(a\_dict)

len(a\_string)

# Dunder Methods

One way that we can introduce polymorphism to our class definitions is by using Python's special dunder methods. We've explored a few already, the constructor \_\_init\_\_ and the string representation method \_\_repr\_\_, but that's only scratching the tip of the iceberg.

Python gives us the power to define dunder methods that define a custom-made class to look and behave like a Python builtin. What does that mean? Say we had a class that has an addition method:

class Color:

def \_\_init\_\_(self, red, blue, green):

self.red = red

self.blue = blue

self.green = green

def \_\_repr\_\_(self):

return "Color with RGB = ({red}, {blue}, {green})".format(red=self.red, blue=self.blue, green=self.green)

def add(self, other):

"""

Adds two RGB colors together

Maximum value is 255

"""

new\_red = min(self.red + other.red, 255)

new\_blue = min(self.blue + other.blue, 255)

new\_green = min(self.green + other.green, 255)

return Color(new\_red, new\_blue, new\_green)

red = Color(255, 0, 0)

blue = Color(0, 255, 0)

magenta = red.add(blue)

print(magenta)

# Prints "Color with RGB = (255, 255, 0)"

In this code we defined a Color class that implements an addition function. Unfortunately, red.add(blue) is a little verbose for something that we have an intuitive symbol for (i.e., the + symbol). Well, Python offers the dunder method \_\_add\_\_ for this very reason! If we rename the add() method above to something that looks like this:

class Color:

def \_\_add\_\_(self, other):

"""

Adds two RGB colors together

Maximum value is 255

"""

new\_red = min(self.red + other.red, 255)

new\_blue = min(self.blue + other.blue, 255)

new\_green = min(self.green + other.green, 255)

return Color(new\_red, new\_blue, new\_green)

Then, if we create the colors:

red = Color(255, 0, 0)

blue = Color(0, 255, 0)

green = Color(0, 0, 255)

We can add them together using the + operator!

# Color with RGB: (255, 255, 0)

magenta = red + blue

# Color with RGB: (0, 255, 255)

cyan = blue + green

# Color with RGB: (255, 0, 255)

yellow = red + green

# Color with RGB: (255, 255, 255)

white = red + blue + green

Since we defined an \_\_add\_\_ method for our Color class, we were able to add these objects together using the + operator.

Instructions

**1.**

In **script.py** there are two classes defined, Atom and Molecule.

Give Atom a .\_\_add\_\_(self, other) method that returns a Molecule with the two Atoms.

**Solution:**

class Atom:

def \_\_init\_\_(self, label):

self.label = label

def \_\_add\_\_(self, other):

return Molecule([self, other])

class Molecule:

def \_\_init\_\_(self, atoms):

if type(atoms) is list:

self.atoms = atoms

sodium = Atom("Na")

chlorine = Atom("Cl")

salt = Molecule([sodium, chlorine])

# salt = sodium + chlorine

# Dunder Methods II

Python offers a whole suite of magic methods a class can implement that will allow us to use the same syntax as Python's built-in data types. You can write functionality that allows custom defined types to behave like lists:

class UserGroup:

def \_\_init\_\_(self, users, permissions):

self.user\_list = users

self.permissions = permissions

def \_\_iter\_\_(self):

return iter(self.user\_list)

def \_\_len\_\_(self):

return len(self.user\_list)

def \_\_contains\_\_(self, user):

return user in self.user\_list

In our UserGroup class above we defined three methods:

* \_\_init\_\_, our constructor, which sets a list of users to the instance variable self.user\_list and sets the group's permissions when we create a new UserGroup.
* \_\_iter\_\_, the iterator, we use the iter() function to turn the list self.user\_list into an iterator so we can use for user in user\_group syntax. For more information on iterators, review [Python's documentation of Iterator Types](https://docs.python.org/3/library/stdtypes.html#typeiter).
* \_\_len\_\_, the length method, so when we call len(user\_group) it will return the length of the underlying self.user\_list list.
* \_\_contains\_\_, the check for containment, allows us to use user in user\_group syntax to check if a User exists in the user\_list we have.

These methods allow UserGroup to act like a list using syntax Python programmers will already be familiar with. If all you need is something to act like a list you could absolutely have used a list, but if you want to bundle some other information (like a group's permissions, for instance) having syntax that allows for list-like operations can be very powerful.

We would be able to use the following code to do this, for example:

class User:

def \_\_init\_\_(self, username):

self.username = username

diana = User('diana')

frank = User('frank')

jenn = User('jenn')

can\_edit = UserGroup([diana, frank], {'can\_edit\_page': True})

can\_delete = UserGroup([diana, jenn], {'can\_delete\_posts': True})

print(len(can\_edit))

# Prints 2

for user in can\_edit:

print(user.username)

# Prints "diana" and "frank"

if frank in can\_delete:

print("Since when do we allow Frank to delete things? Does no one remember when he accidentally deleted the site?")

Above we created a set of users and then added them to UserGroups with specific permissions. Then we used Python built-in functions and syntax to calculate the length of a UserGroup, to iterate through a UserGroup and to check for a User's membership in a UserGroup.

**1.**

In **script.py** you'll find the class LawFirm. Give LawFirm a .\_\_len\_\_() method that will return the number of lawyers in the law firm.

**2.**

Give LawFirm a .\_\_contains\_\_() method that takes two parameters: self and lawyer and checks to see if lawyer is in self.lawyers.

**Solution:**

class LawFirm:

def \_\_init\_\_(self, practice, lawyers):

self.practice = practice

self.lawyers = lawyers

def \_\_len\_\_(self):

return len(self.lawyers)

def \_\_contains\_\_(self, lawyer):

return lawyer in self.lawyers

d\_and\_p = LawFirm("Injury", ["Donelli", "Paderewski"])

# Review

In this lesson, we learned more complicated relationships between classes. We learned:

* How to create a subclass of an existing class.
* How to redefine existing methods of a parent class in a subclass by overriding them.
* How to leverage a parent class's methods in the body of a subclass method using the super() function.
* How to define a Python exception that inherits from Exception.
* How to write programs that are flexible using interfaces and polymorphism.
* How to write data types that look and feel like native data types with dunder methods.

These are really complicated concepts! It's a long journey to get to the state of comfortably being able to build class hierarchies that embody the concerns that your software will need to. Give yourself a pat on the back, you earned it!

Instructions

**1.**

Create a class SortedList that inherits from the built-in type list.

**2.**

Recall that lists have a .append() method that takes a two arguments, self and value. We're going to have SortedList perform a sort after every .append().

Overwrite the append method, leave it blank for now with the pass keyword.

**3.**

First, we want our new .append() to actually add the item to the list.

Write the code that would get SortedList to behave like a normal list when calling the .append() method.

You can call a parent class method using the super() function like this:

def class\_method(self, argument1):

super().class\_method(argument1)

**4.**

After you've appended the new value, sort the list.

You can use the self keyword to refer to the list in question here, so just call self.sort().

**5.**

Incredible! We subclassed a Python primitive and introduced new behavior to it.

Some things to consider:

* When a SortedList gets initialized with unsorted values (say if you call SortedList([4, 1, 5])) those values don't get sorted! How would you change SortedList so that the list is sorted right after the object gets created?
* What other Python builtins have functionality "missing"? Could you write a new dictionary that uses a fallback value when it tries to retrieve an item and can't?

**Solution:**

class SortedList(list):

def append(self, value):

super().append(value)

self.sort()

# Basta Fazoolin'

In this project, you will create classes to store information in your budding restaurant businesses. You will use what you've learned about creating and instantiating classes to describe the relationship between the menu of a restaurant, a restaurant franchise, and a set of businesses in a company.

If you get stuck or confused, remember that your Slack community is there to help!

# Basta Fazoolin'

You've started position as the lead programmer for the family-style Italian restaurant Basta Fazoolin' with My Heart. The restaurant has been doing fantastically and seen a lot of growth lately. You've been hired to keep things organized.

### Making the Menus

1.

At Basta Fazoolin' with my Heart our motto is simple: when you're here with family, that's great! We have four different menus: brunch, early-bird, dinner, and kids.

Create a Menu class .

Remember to use the class keyword when creating a new class.

2.

Give Menu a constructor with the six parameters self, name, items, start\_time, and end\_time.

Remember that a constructor in Python is a method called \_\_init\_\_.

3.

Let's create our first menu: brunch. Brunch is served from 11am to 4pm. The following items are sold during brunch:

{

'pancakes': 7.50, 'waffles': 9.00, 'burger': 11.00, 'home fries': 4.50, 'coffee': 1.50, 'espresso': 3.00, 'tea': 1.00, 'mimosa': 10.50, 'orange juice': 3.50

}

Create a Menu object and save it into the variable brunch. Call it with the arguments "brunch" and the items dictionary in the code block above.

For the start\_time and end\_time, either you can use a 24-hour clock (11 & 16, respectively) or the time object from the [datetime module](https://docs.python.org/3/library/datetime.html#time-objects).

4.

Let's create our second menu item early\_bird. Early-bird Dinners are served from 3pm to 6pm. The following items are available during the early-bird menu:

{

'salumeria plate': 8.00, 'salad and breadsticks (serves 2, no refills)': 14.00, 'pizza with quattro formaggi': 9.00, 'duck ragu': 17.50, 'mushroom ravioli (vegan)': 13.50, 'coffee': 1.50, 'espresso': 3.00,

}

5.

Let's create our third menu, dinner. Dinner is served from 5pm to 11pm. The following items are available for dinner:

{

'crostini with eggplant caponata': 13.00, 'ceaser salad': 16.00, 'pizza with quattro formaggi': 11.00, 'duck ragu': 19.50, 'mushroom ravioli (vegan)': 13.50, 'coffee': 2.00, 'espresso': 3.00,

}

6.

And let's create our last menu, kids. The kids menu is available from 11am until 9pm. The following items are available on the kids menu.

{

'chicken nuggets': 6.50, 'fusilli with wild mushrooms': 12.00, 'apple juice': 3.00

}

7.

Give our Menu class a string representation method that will tell you the name of the menu. Also, indicate in this representation when the menu is available.

Define a string representation like this:

def \_\_repr\_\_(self):

return self.representative\_string

8.

Try out our string representation. If you call print(brunch) it should print out something like the following:

brunch menu available from 11am to 4pm

9.

Give Menu a method .calculate\_bill() that has two parameters: self, and purchased\_items, a list of the names of purchased items.

Have calculate\_bill return the total price of a purchase consisiting of all the items in purchased\_items.

10.

Test out Menu.calculate\_bill(). We have a breakfast order for one order of pancakes, one order of home fries, and one coffee. Pass that into brunch.calculate\_bill() and print out the price.

If you print out

brunch.calculate\_bill(['pancakes', 'home fries', 'coffee'])

You should get the value 13.5.

11.

What about an early-bird purchase? Our last guests ordered the salumeria plate and the vegan mushroom ravioli. Calculate the bill with .caluclate\_bill().

This purchase should result in a bill of 21.5.

### Creating the Franchises

12.

Basta Fazoolin' with my Heart has seen tremendous success with the family market, which is fantastic because when you're at Basta Fazoolin' with my Heart with family, that's great!

We've decided to create more than one restaurant to offer our fantastic menus, services, and ambience around the country.

First, let's create a Franchise class.

13.

Give the Franchise class a constructor. Take in an address, and assign it to self.address. Also take in a list of menus and assign it to self.menus.

14.

Let's create our first two franchises! Our flagship store is located at "1232 West End Road" and our new installment is located at "12 East Mulberry Street". Pass in all four menus along with these addresses to define flagship\_store and new\_installment.

15.

Give our Franchises a string represenation so that we'll be able to tell them apart. If we print out a Franchise it should tell us the address of the restaurant.

Define a .\_\_repr\_\_() method on your Franchise class to give it a string representation.

16.

Let's tell our customers what they can order! Give Franchise an .available\_menus() method that takes in a time parameter and returns a list of the Menu objects that are available at that time.

Write a method .available\_menus() that takes in two parameters, self and time. It should iterate through all of self.menus and return only those menus that are available between start\_time and end\_time.

17.

Let's test out our .available\_menus() method! Call it with 12 noon as an argument and print out the results.

It should return both our brunch menu and our kids menu. Lucky we already gave them string representations!

18.

Let's do another test! If we call .available\_menus() with 5pm as an argument and print out the results.

You should get a list consisitng of the early-bird, the dinner, and the kids menus.

### Creating Businesses!

19.

Since we've been so successful building out a branded chain of restaurants, we've decided to diversify. We're going to create a restaurant that sells arepas!

First let's define a Business class.

20.

Give Business a constructor. A Business needs a name and a list of franchises.

21.

Let's create our first Business. The name is "Basta Fazoolin' with my Heart" and the two franchises are flagship\_store and new\_installment.

22.

Before we create our new business, we'll need a Franchise and before our Franchise we'll need a menu. The items for our Take a' Arepa available from 10am until 8pm are the following:

{

'arepa pabellon': 7.00, 'pernil arepa': 8.50, 'guayanes arepa': 8.00, 'jamon arepa': 7.50

}

Save this to a variable called arepas\_menu.

23.

Next let's create our first Take a' Arepa franchise! Our new restaurant is located at "189 Fitzgerald Avenue". Save the Franchise object to a variable called arepas\_place.

Remember to pass a list consisting of arepas\_menu from the last task.

24.

Now let's make our new Business! The business is called "Take a' Arepa"!

25.

Congrats! You created a system of classes that help structure your code and perform all business requirements you need. Whenever we need a new feature we'll have the well-organized code required to make developing and shipping it easy.

**Solution:**

class menu:

def \_\_init\_\_(self, name, items, start\_time, end\_time):

self.name = name

self.items = items

self.start\_time = start\_time

self.end\_time = end\_time

def \_\_repr\_\_(self):

return "The {name} menu is available from {start\_time} untill {end\_time}.".format(name=self.name,

start\_time=self.start\_time,

end\_time=self.end\_time)

def calculate\_bill(self, purchased\_items):

self.purchased\_items = purchased\_items

price = 0

for purchased\_item in purchased\_items:

if purchased\_item in self.items:

price += self.items[purchased\_item]

return price

brunch = menu("Brunch", {'pancakes': 7.50, 'waffles': 9.00, 'burger': 11.00, 'home fries': 4.50, 'coffee': 1.50, 'espresso': 3.00, 'tea': 1.00, 'mimosa': 10.50, 'orange juice': 3.50}, 11, 16)

early\_bird = menu("Early Bird", {'salumeria plate': 8.00, 'salad and breadsticks (serves 2, no refills)': 14.00, 'pizza with quattro formaggi': 9.00,'duck ragu': 17.50, 'mushroom ravioli (vegan)': 13.50, 'coffee': 1.50, 'espresso': 3.00,}, 15, 18)

dinner = menu("Dinner", {'crostini with eggplant caponata': 13.00, 'ceaser salad': 16.00, 'pizza with quattro formaggi': 11.00,'duck ragu': 19.50, 'mushroom ravioli (vegan)': 13.50, 'coffee': 2.00, 'espresso': 3.00,}, 17, 23)

kids = menu("Kids", {'chicken nuggets': 6.50, 'fusilli with wild mushrooms': 12.00, 'apple juice': 3.00}, 11, 19)

print(brunch.calculate\_bill(['pancakes', 'home fries', 'coffee']))

print(early\_bird.calculate\_bill(['salumeria plate', 'mushroom ravioli (vegan)']))

class Franchise:

def \_\_init\_\_(self, address, menus):

self.address = address

self.menus = menus

def \_\_repr\_\_(self):

return self.address

def available\_menus(self, time):

self.time = time

menus\_available = []

for menu in self.menus:

if time in range(menu.start\_time, menu.end\_time):

menus\_available.append(menu.name)

return menus\_available

flagship\_store = Franchise('1232 West End Road', [brunch, early\_bird, dinner, kids])

new\_installment = Franchise('12 East Mulberry Street', [brunch, early\_bird, dinner, kids])

print(flagship\_store)

print(new\_installment)

print(flagship\_store.available\_menus(12))

print(new\_installment.available\_menus(17))

class Business:

def \_\_init\_\_(self, name, franchises):

self.name = name

self.franchises = franchises

def \_\_repr\_\_(self):

return self.name

first\_business = Business("Basta Fazoolin' with my Heart", [flagship\_store, new\_installment])

arepas\_menu = {'arepa pabellon': 7.00, 'pernil arepa': 8.50, 'guayanes arepa': 8.00, 'jamon arepa': 7.50}

arepas\_place = Franchise("189 Fitzgerald Avenue", arepas\_menu)

take\_a\_arepa = Business("Take a' Arepa", arepas\_place)

print(take\_a\_arepa)

print(first\_business)