INTRODUCTION TO CLASSES

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.

Instructions

1.

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

Hint

You can call `type()` on anything. Like this dictionary!

```
type({'abc': True})
```

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.

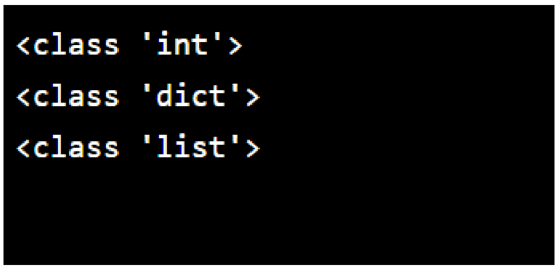
Print out the `type()` of `my_list`.

script.py

```
print(type(5))

my_dict = {}
print(type(my_dict))

my_list = []
print(type(my_list))
```



```
<class 'int'>
<class 'dict'>
<class '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 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!

script.py

```
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 so we can access our instance of `CoolClass` at a later time.

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`.

script.py

```
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

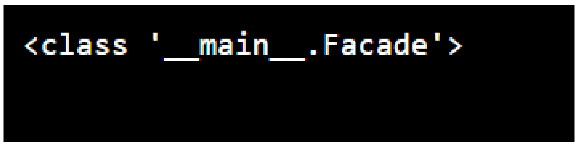
1.
In **script.py** we see `facade_1` from last exercise. Try calling `type()` on `facade_1` and saving it to the variable `facade_1_type`.

2.
Print out `facade_1_type`.

script.py

```
class Facade:
  pass

facade_1 = Facade()
facade_1_type = type(facade_1)
print(facade_1_type)
```



```
<class '__main__.Facade'>
```

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 string "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.

script.py

```
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.

In order for your code to run, you have to have *something* in your class — you can't have a defined class with no body like the following:

```
class exampleClass:
```

For now, make the body of your class `pass`. This will allow your code to run without error.

Hint
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."
```

Since we've now given this class a method, we can remove the `pass` that we added in the previous step.

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

```
class CoolClass:
  def cool_method(self):
    return "Cool!"
```

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

script.py

```
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
```

Hint
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.

Hint
Given a certain diameter, calculate the area using

```
circle.area(diameter / 2)
```

script.py

```
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`.

Instructions

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.

script.py

```python
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)
```

```
New circle with diameter: 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:

```python
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.

```python
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)
```

```python
print(working_string)
# prints "This works! This too!"
```

Instructions

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"`.

script.py

```python
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`.

```python
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? `hasattr()` will return `True` if an object has a given attribute and `False` otherwise. If we want to get the actual value of the attribute, `getattr()` is a Python function that will return the value of a given object and attribute. In this function, we can also supply a third argument that will be the default if the object does not have the given attribute.

The syntax and parameters for these functions look like this:

`hasattr(object, “attribute”)` has two parameters:

- *object* : the object we are testing to see if it has a certain attribute
- *attribute* : name of attribute we want to see if it exists

`getattr(object, “attribute”, default)` has three parameters (one of which is optional):

- *object* : the object whose attribute we want to evaluate
- *attribute* : name of attribute we want to evaluate
- *default* : the value that is returned if the attribute does not exist (note: this parameter is **optional**)

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: a dictionary, a string, an integer, and a list all saved in the variable `can_we_count_it`.

For every element in the list, check if the element has the attribute `count` using the `hasattr()` function. If so, print the following line of code:

```
print(str(type(element)) + " has the count attribute!")
```

Try something like this:

```
for element in list_of_things:
  if hasattr(element, "special_method"):
    # print statment given in instructions
```

2.

Now let’s add an `else` statement for the elements that do not have the attribute `count`. In this `else` statement add the following line of code:

```
print(str(type(element)) + " does not have the count attribute :(")
```

your else statement should look like this:

```
else:
  print(str(type(element)) + " does not have the count attribute :(")
```

Let’s go over the terminal output of the past two instructions. You should see the following output in your terminal right now:

```
<class      'dict'>      does      not      have      the      count      attribute      :(
<class            'str'>             has             the            count            attribute!
<class      'int'>      does      not      have      the      count      attribute      :(
<class 'list'> has the count attribute!
```

This is because dictionaries and integers both do not have a `count` attribute, while strings and lists do. In this exercise, we have iterated through `can_we_count_it` and used `hasattr()` to determine which elements have a `count` attribute. We never actually used the count method, but you can read more about it here if you are curious about what it is.

Click run to move onto the next exercise!

script.py

```
can_we_count_it = [{'s': False}, "sassafrass", 18, ["a", "c", "s", "d", "s"]]

for data_type in can_we_count_it:
  if hasattr(data_type, "count"):
    print(str(type(data_type)) + "has the count attribute!")
  else:
    print(str(type(data_type)) + " does not have the count attribute :(")
<class 'dict'> does not have the count attribute :(
<class 'str'>has the count attribute!
<class 'int'> does not have the count attribute :(
<class 'list'>has the count attribute!
```

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=se
lf.url)

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

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 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
```

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

```
2 * self.pi * self.radius
```

3.
Define three `Circle`s 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.

4.
Print out the circumferences of `medium_pizza`, `teaching_table`, and `round_room`.

script.py

```
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())
```

```
Creating circle with diameter 12
Creating circle with diameter 36
Creating circle with diameter 11460
37.68
113.04
35984.4
```

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.

Hint
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!

script.py

```
print(dir(5))

def this_function_is_an_object():
  pass

print(dir(this_function_is_an_object))
```

```
['__abs__', '__add__', '__and__', '__bool__',
'__ceil__', '__class__', '__delattr__', '__dir__',
'__divmod__', '__doc__', '__eq__', '__float__',
'__floor__', '__floordiv__', '__format__', '__ge__',
'__getattribute__', '__getnewargs__', '__gt__',
'__hash__', '__index__', '__init__',
'__init_subclass__', '__int__', '__invert__',
'__le__', '__lshift__', '__lt__', '__mod__',
'__mul__', '__ne__', '__neg__', '__new__', '__or__',
'__pos__', '__pow__', '__radd__', '__rand__',
'__rdivmod__', '__reduce__', '__reduce_ex__',
'__repr__', '__rfloordiv__', '__rlshift__',
'__rmod__', '__rmul__', '__ror__', '__round__',
'__rpow__', '__rrshift__', '__rshift__', '__rsub__',
'__rtruediv__', '__rxor__', '__setattr__',
'__sizeof__', '__str__', '__sub__',
'__subclasshook__', '__truediv__', '__trunc__',
'__xor__', 'bit_length', 'conjugate', 'denominator',
'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
['__annotations__', '__call__', '__class__',
'__closure__', '__code__', '__defaults__',
'__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__get__',
'__getattribute__', '__globals__', '__gt__',
```

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`.

script.py

```python
class Circle:
  pi = 3.14

  def __init__(self, diameter):
    self.radius = diameter / 2

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

  def area(self):
    return self.pi * self.radius ** 2

  def circumference(self):
    return self.pi * 2 * self.radius


medium_pizza = Circle(12)
teaching_table = Circle(36)
round_room = Circle(11460)

print(medium_pizza)
print(teaching_table)
print(round_room)
```

```
Circle with radius 6.0
Circle with radius 18.0
Circle with radius 5730.0
```

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`.

Hint
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`.

Hint
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.

Hint

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

```
if type(cool_object) is 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!

script.py

```
class Student:
  def __init__(self, name, year):
    self.name = name
    self.year = year
    self.grades = []

  def add_grade(self, grade):
    if type(grade) == Grade:
```

```python
      self.grades.append(grade)

roger = Student("Roger van der Weyden", 10)
sandro = Student("Sandro Botticelli", 12)
pieter = Student("Pieter Bruegel the Elder", 8)

class Grade:
  minimum_passing = 65
  def __init__(self, score):
    self.score = score

grade_1_p = Grade(100)

pieter.add_grade(grade_1_p)
```