8. OOPS with Python

8.a Prospect to Object Oriented Programming in Python

Objectives:

By the end of this chapter, you should be able to:

- Describe OOP without talking about code
- Describe what encapsulation is
- Describe what abstraction is
- Create classes using Python
- Write class and instance methods

In Python 3, everything is an object! We can examine what kind of object using the `type()` function. We've seen examples of this with built-in data types like booleans, strings, and integers. In this chapter, we'll create our own data types, in a way, using *classes*. Any instances of that class that we create will have a type equal to that class.

What is Object Oriented Programming?

Object oriented programming is a method of programming that attempts to model some process or thing in the world as a **class** or **object**. Conceptually, you can think of a class or object as something that has data and can perform operations on that data. With object oriented programming, the goal is to *encapsulate* your code into logical groupings using classes so that you can reason about your code at a higher level. Before we get ahead of ourselves, though, let's define some terminology and see an example:

Class - A blueprint for objects. Classes can contain methods and properties. We commonly use classes to reduce code duplication.

Instance - objects that are made from a class by calling the class. Instances share a similar structure because they all come from the same blueprint (i.e. class).

Poker Example

Say we want to model a game of poker in our program. We could write the program using a list to represent the deck of cards, and then other lists to represent what each player has in their hand. Then we'd have to write a lot of functions to do things like deal, draw cards, see who wins, etc.

When you end up writing large functions and lots of code in one file, there is usually a better way to organize your code. Instead of trying to do everything at once, we could **separate concerns**.

When thinking about a game of poker, some larger processes and objects stand out that you will want to capture in your code:

- Card
- Deck of cards
- Poker hand
- Poker game
- Discard pile (maybe)
- Player
- Bets

Each one of these components could be a class in your program. Let's pick one of the potential classes and figure out the data that it will hold and the functions that it should be able to perform:

Deck of cards

- Cards - the deck should have 52 different playing cards
- Shuffle - the deck should be able to shuffle itself
- Deal a card - remove a card from the deck and deal it to a player
- Deal a hand - The deck may also be used to deal a hand to a player, or a set of players

Now that we can conceptualize how a problem can be broken down into classes, let's talk about why programming this way can be useful.

Encapsulation

Encapsulation is the idea that data and processes on that data are owned by a class. Other functions or classes outside of that class should not be able to directly change the data.

In our deck class, we have 52 cards. The player class should not be able to choose any card he or she wants from the deck or change the order of a deck manually. Instead a player can only be dealt a hand. The contents of the deck is said to be *encapsulated* into the deck class because the deck owns the list of cards and it will not allow other classes to access it directly.

Abstraction

Abstraction is the result of a good object oriented design. Rather than thinking about the details of how a class works internally, you can think about it at a higher level. You can see all of the functions that are made available by the class and understand what the class does without having to see all of the code or worry about implementation details.

Continuing with our example, if you had a deck of cards class and you saw that you could call the `.shuffle()` function or the `.deal()` function, you would have a good understanding of what the class does without having to understand how the functions are working internally.

Other hallmarks of object oriented programming include inheritance and polymorphism. We'll discuss these later.

Creating a class

```
class Vehicle:
    def __init__(self,make,model,year):
        self.make = make
        self.model = model
        self.year = year
```

Every class needs an `__init__` method. Every time you create an instance from a class in Python, that instance will get run through the `__init__`method. `self` inside of this method refers to the instance.

To create an instance from a class, we instantiate the class using () and pass in the values to initialize the instance (these are defined in `__init__`).

```
v = Vehicle('toyota', 'corolla', 2012)
```

Methods and properties for instances

To add methods on instances, we simply define them in the class. When defining instance methods, the first argument should always be called self, and refers to the current instance. If you need to pass other arguments, do so after passing in self. Note that when you call these instance methods, you never pass in self. Here are a couple of examples:

```
class Person():

    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    def full_name(self):
        return f"My name is {self.first_name} {self.last_name}"

    def likes(self, thing):
          return f"{self.first_name} likes {thing}!"

p = Person('Tim', 'Garcia')

p.full_name() # My name is Tim Garcia
p.likes("computers") # Tim likes computers!
```

Notice that we **must** add `self` as the first parameter to each of our instance methods.

Writing Class Methods

We've seen how to add methods on instances. But what if we want to add a method on the class itself? To do this, we use a decorator! We'll talk more about decorators later. For now, it's enough to know that there are two decorators we can use to create class methods:

@classmethod

One way use to use the `@classmethod` decorator:

```
class Person():

    @classmethod
    def say_hello(cls):
        return "HI!"
```

```
Person.say_hello() # "HI!"
```

Similar to instance methods, the first argument in a class method has special meaning. For an instance method, the first argument refers to the instance, and is called `self`. For a class method, the first argument refers to the class, and is typically written as `cls`.

@staticmethod

Another decorator we can use is the `@staticmethod` decorator:

```
class Person():

    @staticmethod
    def say_hello():
        return "HI!"

Person.say_hello() # "HI!"
```

8.b Inheritance and MRO [Method Resolution Order]

Objectives:

By the end of this chapter, you should be able to:

- Explain what inheritance is and how it is used to reduce code duplication
- Understand what the `super` keyword does
- Define MRO and explain how it is used in Python 3

Inheritance and super

In many languages that support object-oriented programming, you can define a class which inherits from another class. Python takes things even further; in Python, we can do what is called multiple inheritance. This means that one class can inherit from many other classes!

Here's an example of single inheritance:

```
class Vehicle:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
    def honk(self):
        return "Beep!"

class Car(Vehicle):
    def __init__(self, make, model, year):
        super().__init__(make, model, year) # this is python3 specific!
        self.wheels = 4
```

Notice that when we define the `Car` class, we pass in the `Vehicle` class. Then, inside of the car's `__init__` method we make a call to `super().__init__`, which refers to the parent class's `__init__` method. Calling `super` in this way ensures that car instances are assigned a `make`, `model`, and `year`, but saves us from having to repeat ourselves in the logic for the `Car`'s `__init__` method.

Inheritance also saves us from having to duplicate instance methods: instances of the child class automatically gain access to instance methods in the parent class. Take a look:

```
mack_truck = Vehicle("Mack", "Titan", 2015)
car = Car("Honda", "Civic", 2004)

mack_truck.honk() # "Beep!"
car.honk() # "Beep!"
```

In this case, we don't need to define a `honk` method for cars. Since the `Car` class inherits from `Vehicle`, any car instances will automatically have access to the `honk` method from the `Vehicle` class.

Multiple Inheritance and MRO

The car and vehicle example is fine when we want a class to inherit from one other class. But what if we want it to inherit from multiple classes? In this case, Python lets us do

multiple inheritance by passing in multiple classes when we create a new class! Here's an example:

```
class Aquatic:
  def __init__(self, name):
    self.name = name

  def swim(self):
    return f"{self.name} is swimming"

  def greet(self):
    return f"I am {self.name} of the sea!"

class Ambulatory:
  def __init__(self, name):
    self.name = name

  def walk(self):
    return f"{self.name} is walking"

  def greet(self):
    return f"I am {self.name} of the land!"

class Penguin(Aquatic, Ambulatory):
  def __init__(self, name):
    super().__init__(name=name)

jaws = Aquatic("Jaws")
lassie = Ambulatory("Lassie")
captain_cook = Penguin("Captain Cook")
```

Here we define two classes: `Aquatic` (for things that can swim) and `Ambulatory` (for things that can walk). Instances of the `Aquatic` class have a `name`, a `swim` method, and a `greet` method. Similarly, instances of the `Ambulatory` class have a `name`, a `walk` method, and a `greet` method.

We then define a third class called `Penguin`, which inherits from both `Aquatic` and `Ambulatory`. Finally, we create an instance from each of these classes.

Let's examine what happens when we try to call methods on each of these instances. You should see the following:

```
jaws.swim() # 'Jaws is swimming'
jaws.walk() # AttributeError: 'Aquatic' object has no attribute 'walk'
jaws.greet() # 'I am Jaws of the sea!'

lassie.swim() # AttributeError: 'Ambulatory' object has no attribute 'swim'
lassie.walk() # 'Lassie is walking'
lassie.greet() # 'I am Lassie of the land!'

captain_cook.swim() # 'Captain Cook is swimming'
captain_cook.walk() # 'Captain Cook is walking'
captain_cook.greet() # 'I am Captain Cook of the sea!'
```

Notice that because `Penguin` inherits from both `Aquatic` and `Ambulatory`, instances of the `Penguin` class have access to both the `swim` method and the `walk` method.

What happens with the `greet` method, though? In this case, there's a conflict, since both `Aquatic` and `Ambulatory` have their own `greet` methods! In this case, it looks like `Penguin` inherits the `greet` method from `Aquatic` and ignores the `greet` method from `Ambulatory`. (Notice that when we defined `Penguin`, we passed in `Aquatic` first, and then `Ambulatory`; what happens if you switch the order?)

So how does Python know where to search for methods? Whenever you create a class, Python sets a **M**ethod **R**esolution **O**rder, or MRO, for that class. This order determines the order in which Python will look for methods on instances of that class. With single inheritance (or even simple examples of multiple inheritance) the order isn't so hard to deduce, but for really complex multiple inheritance things can get tricky. If you ever need to see what the MRO is for a class, you can take a look at the `__mro__` attribute for that class, use the `mro()` method, or, even better, get some `help`:

```
Penguin.__mro__ # (<class 'multiple.Penguin'>, <class 'multiple.Aquatic'>, <c
lass 'multiple.Ambulatory'>, <class 'object'>)

# OR

Penguin.mro() # returns a list to us

# EVEN BETTER!

help(Penguin) # gives us a detailed chain
```

8.c Special Methods & Polymorphism

Objectives:

By the end of this chapter, you should be able to:

- Add special methods to classes
- Define and explain how to implement polymorphism in a class
- Understand how to add custom properties and methods to built in data types in Python

Special methods for python objects

When we discuss special (also called "magic") methods, we are commonly referring to methods that contain a "dunder" (double underscore) like `__init__`. You have actually used quite a few of these under the hood, but you can implement your own for classes you create. Let's see a couple:

`__str__` - what is evaluated when `print` is called

`__len__` - what is evaluated when `len` is called

`__del__` - what is evaluated when `del` is called

`__doc__` - see the docstring for a value

`__class__` - see what class created this object

There also exists another method called `__repr__` which is similar to `__str__`; you can read about the differences here

You can read more about these here. Pay close attention to the differences between `__repr__`, `__str__` and `__format__`

Polymorphism

One of the key principles in Object Oriented Programming is *polymorphism*. This is the idea that an object can take on many (poly) forms (morph).

A very common example you will see of Polymorphism is the `+` operator. If we do `5 + 3` in Python our result will be 8 as the operator knows to add the two numbers. But if we do `"8" + "3"` Python knows to concatenate the values since they are strings. This single operator can take on many different forms, depending on the types it is working with!

You can see polymorphism applied when there is the same operation used for objects in different classes. Let's see another example!

```
sample_list = [1,2,3]
sample_tuple = (1,2,3)
sample_string = "awesome"

len(sample_list)
len(sample_tuple)
len(sample_string)
```

These are three different classes that have the same operation applied to them!

A common implementation of this is to have a method in a base (or parent) class that is overridden by a subclass. Each subclass will have a different implementation of the method. When the superclass/parent class method is called, it chooses which method to run depending on the subclass/child class. Let's look at a common example:

```
class Pet():
    def talk():
        raise NotImplementedError("Subclass needs to implement this method")

class Dog(Pet):
    def talk(self):
        return "WOOF!"

class Cat(Pet):
    def talk(self):
        return "MEOW!"
```

To implement polymorphism you do not **need** a superclass / parent class. It is only when you are using inheritance and polymorphism together that it is useful to ensure that the subclass implements its own version of the method.

For more on polymorphism, check out these articles:

http://blog.thedigitalcatonline.com/blog/2014/08/21/python-3-oop-part-4-polymorphism/#.WBD-PuErLEY

https://pythonspot.com/en/polymorphism/

http://stackoverflow.com/questions/409969/polymorphism-define-in-just-two-sentences

Add custom properties / methods to build in data types in Python

In Python, you can't manipulate base classes for native data types in the same way. However, you can create a new class which extends the built in class, and add methods to the class you've created. To do this, you'll need to import the builtins module and manipulate the class you're interested in.

```
import builtins

# Extended subclass
class extend_str(str):
    def first_last_character(self):
        return self[0] + self[-1]

# Overwrite the str class on builtins with the custom class defined above
builtins.str = extend_str

str(0123).first_last_character() # '03'

str("awesome").first_last_character() # 'ae'

"awesome".first_last_character() # 'str' object has no attribute 'first_last_
character'
```

Note that in the example above, you can't use your custom method on strings unless they're explicitly created by being passed into the `str` class. In short, this approach is cumbersome and not foolproof. But that's probably by design: you shouldn't be trying to modify these classes in the first place!

9. File Operations

9.a File I/O Introduction

Objectives:

By the end of this chapter, you should be able to:

- Read and write to text files
- Explain the purpose of a `with` statement
- Explain the different ways to open a file depending on whether you want to read, write, or append

Definitions

Python has a lot of functionality to help you manipulate text files. In this chapter we'll explore some of this functionality. Before we do so, here are three terms you should know, as we'll be using them frequently when discussing file IO (input/output).

Reading - Getting data from a file. The data can be stored and manipulated in your program.

Writing - Saving new data to a file.

Cursor - When you read a file, a cursor gets created (just like a cursor when you are typing). You can't see this cursor, but this is how you can tell a file where to start reading from. Once the cursor is at the end of a file, you cannot read it anymore unless you explicitly go back to the beginning

Reading

Let's start by creating a text file called `first.txt` and placing the following text inside of it:

```
This is a very simple text file!
```

Now let's make a file called `read.py` and add the following.

```
file = open('first.txt', 'r')
print(file.read())
```

We just opened a file called first.txt for reading (that's what the second argument of 'r' indicates). After opening the file and storing it in a variable called file, we call the read function on it, which allows us to move our cursor from the beginning to the end. Finally, we're not just reading the file; we're printing out the results of that read operation. If you run python3 read.py in Terminal (from the directory where read.py lives), you should see the text of the file in the terminal!

If we try to make invoke file.read() a second time, we just get an empty string back. This is because the cursor is at the end of the file and there is no more to read. In order to read the file again, we'll need to go back to the beginning of the file using seek. Hop into the Python REPL by typing python3 in the terminal, then execute the following code:

```
file = open('first.txt', 'r')

# type in the following commands

file.read() # we see all our test
file.read() # nothing now!

file.seek(0) # move the cursor back to the beginning
file.read() # there it is!
```

But we're not done yet, if we look at file.closed we will see the value is False. So we opened our file, but we never closed it! Let's always make sure we close the file!

```
file = open('first.txt', 'r')

# type in the following commands

file.read() # we see all our test
file.read() # nothing now!

file.seek(0) # move the cursor back to the beginning
file.read() # there it is!

file.closed # False
file.close()
file.closed # True

file.read() # ValueError: I/O operation on closed file
```

Note: for multiline files, you can read one line at a time using the `readline` method.

keyword with

Instead of worrying about closing the file each time, we can use a `with` block, which will automatically close the file after the operation. Let's see what that looks like:

```
with open('first.txt', 'r') as file:
    data = file.read()
    print(data)

file.closed # True
```

Writing

In the previous examples, we've seen that the `open` function takes two parameters. The first is the path to the text file. But what's the second? Here is where we tell Python what we intend to do with the file. Here are the options for what we can pass in:

- `r` use this when you only want to read the file.
- `r+` use this when you want to read and write to the file. In this case, if you write to the file, you'll overwrite any existing characters while you're writing based on the location of the cursor, but once you've finished writing, other characters will be unaffected.
- `a` use this when you want to *append* to the file (text that's already in the file won't be affected)
- `a+` use this when you want to read the file and append to it.
- `w` use this when you want to *write* to the file. In this case, the file is completely emptied before it's written to, so the original text data will be lost.
- `w+` use this when you want to write and read to the file.

If you don't pass anything in to the second argument to `open`, `r` will be assumed.

Take a look at these examples to see how these options behave differently:

```
with open('first.txt', 'w') as file:
    file.write('Lorem ipsum dolor sit amet, consectetur adipisicing elit. Eos
 molestias ea sit veniam, rerum, totam quis eaque excepturi aut, nostrum reic
iendis. At, harum quos adipisci magni nesciunt aliquid beatae sit!')

with open('first.txt', 'r') as file:
    print(file.read())

with open('first.txt', 'r+') as file:
    file.write("\nI'm at the beginning\n")
    file.seek(100)
    file.write("\nI'm in the middle\n")
    file.seek(0)
    print(file.read())

with open('first.txt', 'a+') as file:
    file.write("\nI'm at the end\n")
    file.seek(0)
    print(file.read())

with open('first.txt', 'w+') as file:
    file.write("Now everything is overwritten :(")
    file.seek(0)
    print(file.read())
```

9.b File I/O with CSVs

Objectives:

By the end of this chapter, you should be able to:

- Explain what a CSV file is
- Read and Write to CSV files

CSV with Python

Aside from reading text files, we can also read csv, or comma separated values. CSV files can be opened in a text editor or more commonly in programs like Microsoft Excel. CSVs are a common format for uploading and downloading data, so understanding how to programmatically work with them is important. Here is an example csv file:

`pets.csv`

```
name,type,age
Whiskey,dog,3
Moxie,cat,7
Mascis,dog,2
```

Most languages can actually support files that are separated by any character, not just commas (there is another format called `tsv` for tab separated values). Here is an example of a file that is separated by `|` (we call the character we separate by the "delimiter"):

`file.csv`

```
name|type|age
Whiskey|dog|3
Moxie|cat|7
Mascis|dog|2
```

Now, to read `csv` files, we'll need to import the `csv` module and use the `csv.reader` method. Let's take a look at an example.

Notice that the file we are reading in the example below,`file.csv`, is separated by `|`. Our next step is to read this file using the `csv` module:

```
import csv

with open('file.csv') as csvfile:
    reader = csv.reader(csvfile, delimiter='|')
    rows = list(reader)
    for row in rows:
        print(', '.join(row))

# name, type, age
# Whiskey, dog, 3
# Moxie, cat, 7
# Mascis, dog, 2
```

We can also create a dictionary for each row instead of a list, if we use the `DictReader` method:

```
import csv

with open('file.csv') as csvfile:
    reader = csv.DictReader(csvfile, delimiter='|')
    rows = list(reader)
    for row in rows:
        print(row)

# {'age': '3', 'name': 'Whiskey', 'type': 'dog'}
# {'age': '7', 'name': 'Moxie', 'type': 'cat'}
# {'age': '2', 'name': 'Mascis', 'type': 'dog'}
```

Finally, we can also write to `csv` files in a similar way that we write to plain text files. But rather than creating a `reader`, you'll need to create a `writer`:

```
with open('file.csv', 'a') as csvfile:
    data_writer = csv.writer(csvfile, delimiter="|")
    data_writer.writerow(['Bojack','Horse','50'])
```

As before, you can also write using dictionaries instead of lists. Here's an example for a new `csv` that we're building from scratch.

```
with open('newfile.csv', 'a') as csvfile:
    data = ['name', 'fav_topic']
    writer = csv.DictWriter(csvfile, fieldnames=data)
    writer.writeheader() # this writes the first row with the column headings
    writer.writerow({
        'name': 'Elie',
        'fav_topic': 'Writing to CSVs!'
    })
```