Introduction

Hopefully by now you have learned and somewhat mastered the basics of Python programming. It's now time to move on and learn about *object-oriented programming*.

When using object-oriented programming you write *classes* that represent real world things, you then create *objects* based on these classes. When you create a class you define the general behavior that a whole category of objects can have. When you create objects from a class each object automatically comes with the general behavior of that class. After you create an object you can assign to it any unique behavior that you like.

An example of class could be a book or a car. When you create the class book you can define its general behavior, for example, text size, font-type, color and page type (landscape or portrait). Now that we have the book class created we can create an object from that class. Making an object from a class is called *instantiation*. When you are working with objects, you are working with *instances* of a class.

Learning object-oriented programming is an important step on your Python journey. It will help you write bigger programs that are easier to maintain.

Let's create our first class, the class *ebook*. This class will represent a digital book, not a book for Kindle, or Apple, or a PDF but a general digital book. What do we think is the basic criteria of a digital book? Is should have a name, font-size, font-color, font-type. All of this information will go in our *ebook* class because we can say that they are common to most digital books. Our *ebook* class will instruct Python how to make an object representing an ebook. After we have created our class we'll use it to make instances, each one representing a specific book. Let's go:

In [ ]:

```
1  class DigitalBook:
2      """A class to model a digital book."""
3
4      def __init__(self, name, author):
5          """Initialize name and author attributes."""
6          self.name = name
7          self.author = author
8
9      def open(self):
10         """Simulate a digital book being open in response to an event
11         such as a mouse click or swipe."""
12         print(f"{self.name} is open.")
13
14     def close(self):
15         """Simulate a digital book being closed in response to an event
16         such as a mouse click or swipe."""
17         print(f"{self.name} is closed.")
```

We haven't written code like this before so don't worry if you are a bit confused. Let's go through it.

- At line 1 we define a class called *DigitalBook*. In Python capitalized names refer to classes. You'll also note that there are no parentheses in the class definition because we are creating this class from scratch.
- At line 2 we have a docstring which describes our class.

The init() method

In previous lessons we have created functions. When you write a function inside a class it is called a method. You don't need to relearn anything about functions, don't worry I haven't wasted your time. The big difference between functions that are outside a class and inside, other than being called methods, is how they are called.

The ***init()*** method at line 4 is a special method that Python runs automatically whenever we create a new instance based on the *DigitalBook* class. As you'll have noticed, this method has two leading and two trailing underscores. This helps Python default method names from conflicting with any method names that you may create. If you use only one underscore on each side, then the method won't be called automatically and you'll get unexpected results.

When we define the **init**() method at line 4 we have done so with three paramteres, *self, name,* and *author*. The *self* parameter is required in the method definition and it must come before any other parameter. The reason it must be included in the definition is because when Python calls this method, to create an instance of *DigitalBook*, the method will automatically pass the self argument. Every method call associated with an instance automatically passes *self*, which is a reference to the instance itself, it provides the individual instances access to the attributes and methods in the class.

When we create an instance of *DigitalBook*, Python calls the ***init()*** method from the *DigitalBook* class. As we have added two other parameters after *self*, we will pass a book name and author name as arguments, *self* is passed automatically. Whenever we want to make an instance from the *DigitalBook* class, we need only provide values for the last two parameters.

At lines 6 and 7 we have defined two variables and prefixed them with *self*. A variable prefixed with *self* is available to every method in the class. We'll also be able to access these variables through any instance created from the class.

The code:

```
self.name = name
```

takes the value associated with the parameter *name* and assigns it to the variable *name*, which is then attached to the instance being created. The thing happens with *self.author*. Variables that are accessible through instances like this are called *attributes*.

At lines 9 and 14 we have two other methods defined, *open()* and *close()*. These methods don't need any additional information to run so we just define them to have one parameter, *self*. Any instances we created later will have access to these methods. This means that they will be able to open and close. At the moment *open()* and *close()* don't do too much, they simply print a message telling a user if the book is open of closed. But think about it in the context of real digital book on a tablet or mobile device. What we have just written would be used by someone to see the name of a book in their library and if they are reading it or not.

Making an instance from a class

We now have our first class *DigitalBook*, which is a set of instructions for how to make an individual instance of that class representing a specific digital book.

Now, let's create an instance representing a specific digital book:

In [ ]:

```
# This code can be added after the last line of the DigitalBook class

my_book = DigitalBook('Learning Python', 'tony staunton')
print(f"The book I am reading is called {my_book.name}.")
print(f"It was written by {my_book.author.title()}.")
```

In this piece of code at line 3 we are telling Python to create a digital book called 'Learning Python' by the author 'Tony Staunton'. When Python reads this code it calls the **init**() method in *DigitalBook* with the arguments 'Learning Python' and 'Tony Staunton'. The **init**() method creates an instance representing this particular digital book and set the name and author attributes using the values we provided at line 3. The instance that is created is assigned to the variable *my_book*.

Accessing Attributes

Remember what attributes are? Variables that are accessible through instances are called attributes. To access the attributes of an instance we can use dot notation. At line 4 in the code block above we access the attribute *name* of *my_book* with the code:

```
my_book.name
```

You find dot notation frequently in Python so it's good to be able to recognize it. The syntax at line 4, *my_book.name* shows us how Python finds an attribute's value. Python looks at the instance *my_book* and then finds the attribute *name* associated with it. Remember, this is the same attribute that we referred to as *self.name* in the class *DigitalBook*. At line 5 in the above code block we have used the same approach to get the *author* attribute only this time we have used Python's *title* method.

In [ ]:

```
my_book = DigitalBook('Learning Python', 'tony staunton')
print(f"The book I am reading is called {my_book.name}.")
print(f"It was written by {my_book.author.title()}.")
```

Calling methods

Do you remember what a *method* is? A method is any function that is created inside a class. Now that we have created an instance from the class *DigitalBook* called *my_book*, we can again use dot notation to call any method defind in *DigitalBook*:

In [ ]:

```
my_book.open()
```

In [ ]:

```
my_book.close()
```

To call a method in Python you provide the name of the instance, which in this example is *my_book* and the method that you want to call, separated by a dot. Python will read *my_book.open()* and look for the method *open()* in the class *DigitalBook* and run that code.

As you have probably noticed reading this code, and calling methods using dot notation becomes easier the more you do it. Giving our methods helpful, descriptive names makes it easier for us and other developers to know what a block of code does.

Creating multiple instances

SO far we have created only one instance from the class *DigitalBook* Let's change that by creating a second book:

In [ ]:

```
your_book = DigitalBook('Atomic Habits', 'James Clear')

print(f"\nYou are reading {your_book.name}.")
print(f"\nIt was written by {your_book.author}.")
```

In this code example we have created a second instance of the class *DigitalBook*. Each book is a separate instance with its own set of attributes and has the same functionality:

In [ ]:

```
your_book.open()
```

In [ ]:

```
your_book.close()
```

We could have created this second instance with the same name and author values as we used in the first instance of *DigitalBook*. Python still creates a separate instance from the *DigitalBook* class. You can create as many instances as you need from one class so long as you give each instance a unique variable name (my_book, your_book) or it has a unique spot in a list or dictionary.

Exercises

- Create a new class called *Car*. The **init**() method for *Car* should store two attributes, a car name (ford) and a car type (manual or automatic).
- Create a method called *car_order()* that prints the two attributes and another method called *order_status* that confirms a car has been ordered.
- Make an instance called *car* from your class. print the two attributes individually and then call both methods.

Working with classes and instances

As mentioned, classes can used to simulate real-world situations. After you have created a class, most of your time as a developer will be spent working with instances created from a class. One of the first things that you will need to do when working with instances of a class is know how to modify the attributes associated with a particular instance. Attributes of an instances can be modified directly or you can write methods that update attributes in specific ways. Let's take a look:

The tablet class

Let's write a new class representing a tablet such as an iPad. Our new class will store information about the kind of tablet we're working with, and it will have a method that summarizes this information:

In [ ]:

```
1  class Tablet:
2      """A simple representation of a tablet."""
3
4      def __init__(self, make, model, screen_size):
5          """Initialize attributes to describe a tablet."""
6          self.make = make
7          self.model = model
8          self.screen_size = screen_size
9
10     def get_tablet_description(self):
11         """Return a tablet description."""
12         description = f"{self.make}, {self.model}, {self.screen_size}"
13         return description
14
15 new_tablet = Tablet('ipad', 'Pro', '12.3 inches')
16 print(new_tablet.get_tablet_description())
```

In this code example we create a new class called *Tablet* at line 1. The **init**() method at line 4, has 4 parameters. First there is the *self* parameter, followed by *make*, *model*, and *screen_size*. The **init**() method takes these parameters and assigns them to the attributes that will be associated with instances made from this class. When we create a new *Tablet* instance, we need to specify a make, model and screen size for our instance.

Our second method defined at line 10 is called *get_tablet_description*. This method places the make, model and screen size of a tablet into one string of text. At line 15 we make a new instance from the *Tablet* class and assign it to the variable *new_tablet*. We then call *get_tablet_description()* to view the output.

Setting a default value for an attribute

When an instance is created, attributes can be defined without being passed in as parameters. These attributes can be defined in the ***init()*** method where they are assigned a default value.

Let's update out *Tablet* class with a new attribute called *total_storage* with a value of 256GB. We'll also add a method called *available_storage* that will read out the memory of each tablet.

In [ ]:

```
class Tablet:
    """A simple representation of a tablet."""

    def __init__(self, make, model, screen_size):
        """Initialize attributes to describe a tablet."""
        self.make = make
        self.model = model
        self.screen_size = screen_size
        self.storage_remaining = 256

    def get_tablet_description(self):
        """Return a tabler description."""
        description = f"{self.make}, {self.model}, {self.screen_size}"
        return description

    def available_storage(self):
        """Print a message display a tablets available storage."""
        print(f"This device has {self.storage_remaining} GB available of 256 GB.")

new_tablet = Tablet('ipad', 'Pro', '12.3 inches')
print(new_tablet.get_tablet_description())
new_tablet.available_storage()
```

Now, when Python calls the ***init()*** method to create a new instance, it stores the make, model, and screen_size values as attributes just as we did in the previous code example. At line 9 Python creates a new attribute called *total_storage* and sets its initial value to 256. At line 16 we have created a new method called *available_storage* which prints out the available storage for a tablet.

As we all know the memory in our devices never lasts very long so we need a way to change the value of the attribute *storage_remaining*.

Modifying an attributes value

There are 3 ways to change an attributes value:

1. Change the value directly through an instance
2. Set the value through a method
3. Increment the value through a method

Let's take a look at each of these in turn.

Modifying an attributes value directly

This is the easiest way to modify the value of an attribute. We access an attribute directly through an instance. Let's revisit our tablet class and update *storage_remaining* to 212.

In [ ]:

```
class Tablet:
    """A simple representation of a tablet."""

    def __init__(self, make, model, screen_size):
        """Initialize attributes to describe a tablet."""
        self.make = make
        self.model = model
        self.screen_size = screen_size
        self.storage_remaining = 256

    def get_tablet_description(self):
        """Return a tabler description."""
        description = f"{self.make}, {self.model}, {self.screen_size}"
        return description

    def available_storage(self):
        """Print a message display a tablets available storage."""
        print(f"This device has {self.storage_remaining} GB available of 256 GB.")

new_tablet = Tablet('ipad', 'Pro', '12.3 inches')
print(new_tablet.get_tablet_description())

# Modifying an attributes value
new_tablet.storage_remaining = 212
new_tablet.available_storage()
```

In this code example we have use dot notation to access the tablet's *storage_remaining* attribute and set its value directly. At line 24 we tell Python to take the new instance *new_tablet*, find the attribute *storage_remaining* and set its value to 212.

How about writing a method that updates the value of an attribute for us? Let's take a look:

Modifying an attributes value through a method

Instead of accessing and updating an attribute directly as we did in the previous code example we can pass a new value to a method that will do the updating for us.

In [ ]:

```
class Tablet:
    """A simple representation of a tablet."""

    def __init__(self, make, model, screen_size):
        """Initialize attributes to describe a tablet."""
        self.make = make
        self.model = model
        self.screen_size = screen_size
        self.storage_remaining = 256

    def get_tablet_description(self):
        """Return a tabler description."""
        description = f"{self.make}, {self.model}, {self.screen_size}"
        return description

    def available_storage(self):
        """Print a message display a tablets available storage."""
        print(f"This device has {self.storage_remaining} GB available of 256 GB.")

    def update_storage(self, storage):
        """Set the available storage to the value given."""
        self.storage_remaining = storage

new_tablet = Tablet('ipad', 'Pro', '12.3 inches')
print(new_tablet.get_tablet_description())

# Modifying an attributes value
new_tablet.update_storage(158)
new_tablet.available_storage()
```

In this code example we have added a new method called *update_storage()* at line 20. This method receives a storage value and assigns it to *self.storage_remaining*. At line 28 we call the *update_storage()* method and pass it *158* as an argument. It sets the storage remaining to 158 and the method *available_storage* prints it out.

What happens when we try to install an application on out tablet that is greater than the storage currently available? At the moment, nothing! But let's fix that!

In [ ]:

```
class Tablet:
    """A simple representation of a tablet."""

    def __init__(self, make, model, screen_size):
        """Initialize attributes to describe a tablet."""
        self.make = make
        self.model = model
        self.screen_size = screen_size
        self.storage_remaining = 256

    def get_tablet_description(self):
        """Return a tablet description."""
        description = f"{self.make}, {self.model}, {self.screen_size}"
        return description

    def available_storage(self):
        """Print a message display a tablets available storage."""
        print(f"This device has {self.storage_remaining} GB available of 256 GB.")

    def update_storage(self, storage):
        """
        Set the available storage to the value given.
        Reject the change if storage amount is less than storage remaining.
        """
        if storage < self.storage_remaining:
            self.storage_remaining = storage
        else:
            print("There is not enough storage available")

new_tablet = Tablet('ipad', 'Pro', '12.3 inches')
print(new_tablet.get_tablet_description())

# Modifying an attributes value
new_tablet.update_storage(300)
new_tablet.available_storage()
```

In this updated code *update_storage* checks that the new storage value is possible before modifying the attribute.

Incrementing an attributes value through a method

There will be times when you want to increment an attributes value by a certain amount instead of setting an entirely new value. When might this happen? For example, if you have 158GB storage remaining on your tablet and you remove an application that is occupying 4GB of space then you would need to increment 158 by 4 to 162. Let's take a look:

In [ ]:

```
class Tablet:
    """A simple representation of a tablet, such as an iPad."""

    def __init__(self, make, model, screen_size):
        """Initialize attributes to describe a tablet."""
        self.make = make
        self.model = model
        self.screen_size = screen_size
        self.total_storage = 256

    def get_tablet_description(self):
        """Return a description of a tablet."""
        tablet_description = f"{self.make}, {self.model}, {self.screen_size}"
        return tablet_description

    def available_storage(self):
        """Print a message to display a tablets available storage."""
        print(f"This tablet has a storage capacity of {self.total_storage} GB.")

    def update_storage(self, storage):
        """
        Set the available storage to the value given.
        Reject the change if storage amount is less than total storage.
        """
        if storage < self.total_storage:
            self.total_storage = self.total_storage - storage
        else:
            print("There is not enough storage available.")

    def increment_total_storage(self, application):
        """Add the amount removed to total storage available."""
        self.total_storage += application

new_tablet = Tablet('ipad', 'pro', '12.3 inches')
print(new_tablet.get_tablet_description())

new_tablet.update_storage(36)
new_tablet.available_storage()

new_tablet.increment_total_storage(80)
new_tablet.available_storage()

```

In this code example we have added a new method at line 30, *increment_total_storage*. This method receives an amount of storage and adds it to the value of *total_storage*.

At line 34 we create a new tablet. At line 37 we reduce the storage of our tablet by calling *update_storage()* and passing it 36. Then at line 40 we call *increment_total_storage* and pass it 80 which is added to the 220 GB of total storage.

Exercises

- Start with your class from the previous exercise, Car. Add an attribute called odometer and set the default value to 100, meaning that the car has traveled 100 miles.
- Create a new instance from the Car class and print out the odometer reading.
- Add a new method called *increment_odometer()* that lets you increment the odometer value as if the car had traveled on a journey of 550 miles.
- Print out the new value of the odometer.
- Odometers cannot go backwards in value, update the *increment_odometer()* method to print an error message if an increment is less than the current value of the odometer.

Inheritance

In the code we have written so far for our classes we have written everything from scratch. But you don't always have to start from 0. If a class you are writing is a version of another class you have written then you can use *inheritance*. When one class *inherits* from another it takes on the attributes and methods of the first class. In this scenario the original class is called the *parent class* and the new class is called the *child class*. The child class can inherit any or all of the attributes and methods of its parent class. But, it can also define new attributes and methods of its own.

The init() method for a child class

When you're creating a new class that is based on an existing class you will often want to call the **init**() method from the parent class. Why? Because this will initialize any attributes that were defined in the parent **init**() method and make them available in the child class.

Let's demonstrate this by creating a new class Ereader class that will do everything that our Tablet class does:

In [ ]:

```
1  class Tablet:
2      """A simple representation of a tablet, such as an iPad."""
3
4      def __init__(self, make, model, screen_size):
5          """Initialize attributes to describe a tablet."""
6          self.make = make
7          self.model = model
8          self.screen_size = screen_size
9          self.total_storage = 256
10
11     def get_tablet_description(self):
12         """Return a description of a tablet."""
13         tablet_description = f"{self.make}, {self.model}, {self.screen_size}"
14         return tablet_description
15
16     def available_storage(self):
17         """Print a message to display a tablets available storage."""
18         print(f"This tablet has a storage capacity of {self.total_storage} GB.")
19
20     def update_storage(self, storage):
21         """
22         Set the available storage to the value given.
23         Reject the change if storage amount is less than total storage.
24         """
25         if storage < self.total_storage:
26             self.total_storage = self.total_storage - storage
27         else:
28             print("There is not enough storage available.")
29
30     def increment_total_storage(self, application):
31         """Add the amount removed to total storage available."""
32         self.total_storage += application
33
34 class Ereader(Tablet):
35     """Represent aspects of a tablet specific to an ereading device."""
36
37     def __init__(self, make, model, screen_size):
38         """Initialize attributes of the parent class."""
39         super().__init__(make, model, screen_size)
40
41 my_ereader = Ereader('kindle', 'paper weight', '6 inches')
42 print(my_ereader.get_tablet_description())
```

In this code example we have our *Tablet* class starting at line 1. When you are writing a child class, the parent class must be part of the current file and it must appear before the child class in the file.

At line 34 we define our child class called *Ereader*. As you can see we have included the name of the parent class in parentheses in the definition of the child class, this is a requirement when creating a child class.

The **init**() method at line 37 takes the information required to make a *Tablet* instance.

At line 39 we have the *super()* function. This is a special function that allows us to call a method from the parent class. This function tells Python to call the **init**() method from *Tablet*, which gives an instance of *Ereader* all the attributes defined in that method. The name *super* comes from a programming convention of calling the parent class a *superclass* and a child class a *subclass*.

At line 41 we create an ereader with the same kind of information we'd provide when creating a tablet. We make an instance of the *Ereader* class and assign it to *my_ereader*. Line 41 calls the **init**() method defined in *Ereader* which in turn tells Python to call the **init**() method defined in the parent class *Tablet*.

Apart from the **init**() method we haven't added any additional attributes or methods.

As you can see from the output our new *Ereader* class is working as expected. Now we can begin to define attributes and methods unique to ereaders.

Defining attributes and methods for a child class

Now, let's add an attribute that is specific to ereaders, for example a library, and a method to report how many books the ereader can hold.

In [ ]:

```
class Tablet:
    """A simple representation of a tablet, such as an iPad."""

    def __init__(self, make, model, screen_size):
        """Initialize attributes to describe a tablet."""
        self.make = make
        self.model = model
        self.screen_size = screen_size
        self.total_storage = 256

    def get_tablet_description(self):
        """Return a description of a tablet."""
        tablet_description = f"{self.make}, {self.model}, {self.screen_size}"
        return tablet_description

    def available_storage(self):
        """Print a message to display a tablets available storage."""
        print(f"This tablet has a storage capacity of {self.total_storage} GB.")

    def update_storage(self, storage):
        """
        Set the available storage to the value given.
        Reject the change if storage amount is less than total storage.
        """
        if storage < self.total_storage:
            self.total_storage = self.total_storage - storage
        else:
            print("There is not enough storage available.")

    def increment_total_storage(self, application):
        """Add the amount removed to total storage available."""
        self.total_storage += application

class Ereader(Tablet):
    """Represent aspects of a tablet specific to an ereading device."""

    def __init__(self, make, model, screen_size):
        """
        Initialize attributes of the parent class.
        Then initialize attributes specific to an ereader.
        """

        super().__init__(make, model, screen_size)
        self.library_size = 1000

    def describe_library(self):
        """Print a message describing the library size."""
        print(f"This ereader can hold {self.library_size} books.")

my_ereader = Ereader('kindle', 'paper weight', '6 inches')
print(my_ereader.get_tablet_description())
my_ereader.describe_library()
```

At line 44 we add a new attribute to the *Ereader* class, *self.library_size*, and we set its initial value to 1000. Because this attribute is defined in the child class, *Ereader* it will be associated with all instances created from that class. It won't be associated with any instances of the *Tablet* class.

At line 46 we added a new method called *describe_library()*. This method prints out a message telling us how many books the ereader can hold in its library. The output is now clearly specific to an ereading device.

You can add as many attributes as you want to the *Ereader* class to make it even more specific. When you are adding new attributes and methods to either class be mindful or where they should live. An attribute or method that describes a general tablet should go in the tablet class and an attribute or method that is more specific to an ereader should go in the ereader class.

Overriding methods from a parent class

At some stage in life all children rebel against their parents and so it is in programming as well. In Python you have the ability to override any method from the parent class that may not be appropriate for what you are trying to achieve with the child class. To do this, you first define a method in the child class with the same name as the method you want to override in the parent class. Python will ignore the parent class method and only use the method you define in the child class.

Imagine our *Tablet* class had a method called *pencil_battery_status*. We don't need this method in our *Ereader* class so we could override it with something like:

In [ ]:

```
class Ereader(Tablet):

    def pencil_battery_status(self):
        """Ereaders don't have pencils."""
        print("No pencil is required with this device.")
```

If a programmer tries to call *pencil_battery_status* within the Ereader class, Python will ignore the method in *Tablet* and instead run this method.

Instances as attributes

As you begin to extend your class to do more and represent more complex items from the real world you'll find that you files start to become bigger and bigger. When you notice this start to happening it is worth reviewing classes to determine if anything can be written as a separate class. Large classes can be broken up into smaller ones that all work together.

If we were to keep adding attributes and methods to our *Ereader* class, we might start to notice that we are adding lots of details around one particular object or area, a battery for example. In this case we could move attributes and methods related to a battery into a separate class called *Battery*. We could then use a *Battery* instance as an attribute in the *Ereader* class. Let's take a look:

In [ ]:

```
1  class Tablet:
2      """A simple representation of a tablet, such as an iPad."""
3
4      def __init__(self, make, model, screen_size):
5          """Initialize attributes to describe a tablet."""
6          self.make = make
7          self.model = model
8          self.screen_size = screen_size
9          self.total_storage = 256
10
11     def get_tablet_description(self):
12         """Return a description of a tablet."""
13         tablet_description = f"{self.make}, {self.model}, {self.screen_size}"
14         return tablet_description
15
16     def available_storage(self):
17         """Print a message to display a tablets available storage."""
18         print(f"This tablet has a storage capacity of {self.total_storage} GB.")
19
20     def update_storage(self, storage):
21         """
22         Set the available storage to the value given.
23         Reject the change if storage amount is less than total storage.
24         """
25         if storage < self.total_storage:
26             self.total_storage = self.total_storage - storage
27         else:
28             print("There is not enough storage available.")
29
30     def increment_total_storage(self, application):
31         """Add the amount removed to total storage available."""
32         self.total_storage += application
33
34 class Battery:
35     """A simple attempt to represent the battery of an ereader."""
36
37     def __init__(self, battery_size = 1500):
38         """Initialize the battery's attributes."""
39         self.battery_size = battery_size
40
41     def describe_battery(self):
42         """Print a message describing the size of the battery."""
43         print(f"This ereader has a battery size of {self.battery_size} mAh.")
44
45 class Ereader(Tablet):
46     """Represent aspects of a tablet specific to an ereading device."""
47
48     def __init__(self, make, model, screen_size):
49         """
50         Initialize attributes of the parent class.
51         Then initialize attributes specific to an ereader.
52         """
53
54         super().__init__(make, model, screen_size)
55         self.library_size = 1000
56         self.battery = Battery()
57
58     def describe_library(self):
59         """Print a message describing the library size."""
60         print(f"This ereader can hold {self.library_size} books.")
61
62 my_ereader = Ereader('kindle', 'paperwhite', '6 inches')
63
64 print(my_ereader.get_tablet_description())
65 my_ereader.battery.describe_battery()
```

In this code example we create a new class at line 34 called *Battery*. This class does not inherit from any other class. The **init**() method at line 37 has only one parameter, apart from self, *battery_size*. This parameter is optional and it sets the size of the battery to 1500 if no value is provided.

At line 56 in the *Ereader* class we have added an attribute called *self.battery*. Python will create a new instance of *Battery* with a default value of 1500 as we have not specified any other value and assign the that instance to *self.battery*. This will happen every time the **init**() method is called and any *Ereader* instance will have a *Battery* instance created automatically.

At line 62 we create an Ereader and assign it to the variable *my_ereader*. At line 65 we are working with our new battery. This line of code is telling Python to look at the instance of *my_ereader*, find its *battery* attribute and then call the method *describe_battery()* that is associated with the *Battery* instance stored in the attribute.

This may seem like a lot of extra work but this approach allows us to add as much detail as we want to the battery without adding code bloat to the *Ereader* class.

Let's add another method to our *Battery*, this method will inform a user how much charge there is in the battery.

In [ ]:

```
1  class Tablet:
2      """A simple representation of a tablet, such as an iPad."""
3
4      def __init__(self, make, model, screen_size):
5          """Initialize attributes to describe a tablet."""
6          self.make = make
7          self.model = model
8          self.screen_size = screen_size
9          self.total_storage = 256
10
11     def get_tablet_description(self):
12         """Return a description of a tablet."""
13         tablet_description = f"{self.make}, {self.model}, {self.screen_size}"
14         return tablet_description
15
16     def available_storage(self):
17         """Print a message to display a tablets available storage."""
18         print(f"This tablet has a storage capacity of {self.total_storage} GB.")
19
20     def update_storage(self, storage):
21         """
22         Set the available storage to the value given.
23         Reject the change if storage amount is less than total storage.
24         """
25         if storage < self.total_storage:
26             self.total_storage = self.total_storage - storage
27         else:
28             print("There is not enough storage available.")
29
30     def increment_total_storage(self, application):
31         """Add the amount removed to total storage available."""
32         self.total_storage += application
33
34 class Battery:
35     """A simple attempt to represent the battery of an ereader."""
36
37     def __init__(self, battery_size = 1500):
38         """Initialize the battery's attributes."""
39         self.battery_size = battery_size
40
41     def describe_battery(self):
42         """Print a message describing the size of the battery."""
43         print(f"This ereader has a battery size of {self.battery_size} mAh.")
44
45     def display_batttery_charge(self):
46         """Print a statement informing a user how much charge is left in the current battery"""
47         if self.battery_size == 1500:
48             life = 10
49         elif self.battery_size ==1000:
50             life = 8
51
52         print(f"There are {life} hours of battery remaining.")
53
54 class Ereader(Tablet):
55     """Represent aspects of a tablet specific to an ereading device."""
56
57     def __init__(self, make, model, screen_size):
58         """
59         Initialize attributes of the parent class.
60         Then initialize attributes specific to an ereader.
61         """
62
63         super().__init__(make, model, screen_size)
64         self.library_size = 1000
65         self.battery = Battery()
66
67     def describe_library(self):
68         """Print a message describing the library size."""
69         print(f"This ereader can hold {self.library_size} books.")
70
71
72 my_ereader = Ereader('kindle', 'paperwhite', '6 inches')
73
74 print(my_ereader.get_tablet_description())
75 my_ereader.battery.describe_battery()
76 my_ereader.battery.display_batttery_charge()
```

The new method, *display_battery_charge* added at line 45 calculates how much usage time the ereader has based on the size of the battery. If the battery size is 1500 then there is 10 hours of ereader usage in the battery. If the battery is 1000 then there is 8 hours of usage in the battery.

When we want to call this method we do so through the ereader's *battery* attribute.

Modeling real-world objects, and logical thinking over syntax

Exercises

- Staying with our ereader class find the *describe_library* method and move this to a class of its own within the *Tablet* class.
- Expand the *library* class to display how may books are in the ereader library, book genres, and how many books have been read. Add new attributes for each of these
- Write methods that display number of books in the library, book genres, and how many book have been read.

Importing classes

As you can see from the programs we have been writing recently, they can get big very quickly and without you realizing it. When this starts to happen you'll want to move your classes into modules and then import the classes you need into your programs. This will help you to keep your files as uncluttered and clean as possible.

Importing from a single class

For this next code example we will create a module that contains only our tablet class. Be careful when creating new modules based on your current code as you may already have files with similar names.

At line one of our new module we have added a docstring that gives a description of what the module does. It is best practice to write a docstring at the beginning of your modules.

Now lets write the code to import the *Tablet* class and then create an instance from that class.

In [ ]:

```
1 from tablet import Tablet
2
3 new_tablet = Tablet('iPad', 'Air', '10.3 inch')
4 print(new_tablet.get_tablet_description())
5
6 new_tablet.update_storage(1000)
7 new_tablet.available_storage()
```

The import statement at line 1 tells Python to open the *tablet* module and import the class *Tablet*. This allows us to use the *Tablet* class as if we had written it inside of this piece of code. The output is no different than in previous code examples.

Importing classes in this way is a great, clean, clutter free way to write programs.

Storing multiple classes in a module

We can store as many classes as we need in a single module. However, it is best practice to ensure that each class in a module is related. Let's add our *battery* and *ereader* classes to *tablet.py*.

Now let's write the code to import the *Ereader* class and create an ereader:

In [ ]:

```
1 from tablet import Ereader
2
3 new_ereader = Ereader('kindle', 'paper weight', '6 inches')
4
5 print(new_ereader.get_tablet_description())
6 new_ereader.describe_library()
```

The output is the same as the before.

Importing multiple classes from a Module

We can import as many class as we need into our programs. If we wanted to create an ereader and an iPad in the same file we could import both classes. Take a look:

In [12]:

```
1 from tablet import Tablet, Ereader
2
3 kindle = Ereader('kindle', 'paper weight', '6 inches')
4 print(kindle.get_tablet_description())
5
6 ipad = Tablet('iPad', 'Air', '10.3 inch')
7 print(ipad.get_tablet_description())
```

```
kindle, paper weight, 6 inches
iPad, Air, 10.3 inch
```

Here we have combined the previous two code examples into one file. At line one we have an import statement and we have added two classes separated by a comma.

Importing an entire module

We can import an entire module and then access any class we need using dot notation. This is a simple approach to importing. Take a look:

In [13]:

```
1 import tablet
2
3 kindle = tablet.Ereader('kindle', 'paper weight', '6 inches')
4 print(kindle.get_tablet_description())
5
6 ipad = tablet.Tablet('iPad', 'Air', '10.3 inch')
7 print(ipad.get_tablet_description())
```

```
kindle, paper weight, 6 inches
iPad, Air, 10.3 inch
```

As you can see the statement at line 1 now reads "import tablet", which import the entire module.

You can also see at lines 3 and 6 that we have used dot notation to access the classes, *Ereader* and *Tablet*.

Import all class from a module

We can import every class from a module with the code:

In [14]:

```
1 from tablet import *
2
3 kindle = Ereader('kindle', 'paper weight', '6 inches')
4 print(kindle.get_tablet_description())
5
6 ipad = Tablet('iPad', 'Air', '10.3 inch')
7 print(ipad.get_tablet_description())
```

```
kindle, paper weight, 6 inches
iPad, Air, 10.3 inch
```

However this approach is not recommended for the following reasons:

1. It is helpful to read the import statements at the top of a file to understand exactly what a program is importing and what classes are being used.
2. This approach can also lead to confusion in file naming. What happens you import a class with a name that is already in use in your program?

The best practice approach is to import an entire module and then using the *module_name.ClassName* syntax.

Importing a module into a module

If your Python files start to grow extremely big then you can start to spread your classes out over several modules. Let's store the *Tablet* class in one module and the *Ereader* and *Battery* classes in a separate module.

Create a new module called *ereader.py* and place the *Ereader* and *Battery* classes in there.

- see ereader.py file

The class *Ereader* needs access to its parent class and so we import *Tablet* directly into the module at line 1. Next we need to update our *Tablet* module so that it contains only one class.

Now we can import from each module separately.

In [10]:

```
from tablet import Tablet
from ereader import Ereader

kindle = Ereader('kindle', 'paper weight', '6 inches')
print(kindle.get_tablet_description())

ipad = Tablet('iPad', 'Air', '10.3 inch')
print(ipad.get_tablet_description())
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Input In [10], in <cell line: 2>()
      1 from tablet import Tablet
----> 2 from ereader import Ereader
      4 kindle = Ereader('kindle', 'paper weight', '6 inches')
      5 print(kindle.get_tablet_description())

ModuleNotFoundError: No module named 'ereader'
```

In this code example we import *Tablet* from its module and *Ereader* from its module.

Using Aliases

We've seen aliases before and they are great at helping to make out code more compact. Aliases can also be used when importing classes as well.

```
from ereader import Ereader as ER

kindle = ER('kindle', 'paper weight', '6 inches')
```

This alias can be used whenever we need to make an ereader.

Finding your own approach

Python gives us a lot of ways to work with and organize large code files. It's good to know all of the options that you have so that you can decide on which approach works best for you.

All of these approaches may seem a little overwhelming and the best advice that I can give you is to keep it simple. Don't over complicate Unnecessarily. Keep your code in one file and when everything is working as you expect then start to move things around if you need to.

Exercises

- Break the *Tablet*, *Ereader*, and *Battery* classes into seperate modules and import one by one into a separate python file called new_tablet.py. Create a new instance and make sure your classes are working as expected.

The Python standard library

Now that you know how classes and modules work and how to import them it's time to learn about the Python standard library. This is a set of pre-written modules included with every Python installation. You can start to use these modules in your own programs which will save you from writing your own.

You can use any function or classes from the Python Standard Library by including an import statement at the top of your file. Let's take a look:

In [ ]:

```
from random import randint
randint(1,10)
```

In [ ]:

```
from random import choice
quotes = ['quote1', 'quote2', 'quote3']
daily_quote = choice(quotes)
daily_quote
```

Exercises

- Go explore other functions and classes from the Python Standard Library