LAB04-01_code-reuse (ASSIGN04)

November 2, 2021

Practice with Code Reuse in Python

Go through the self-learning text below, fill in the missing code in the exercises and submit this notebook as part of your Assignment 3

This notebook is used both as a lecture and as an assignment. I'll illustrate some important concepts during the lecture. Then, as an assignment, you´ll go through this notebook again, fill in the missing code snippets and submit it as part of your Assignment 3. There are many ways in which we can reuse code. We have seen how **functions** and **classes** can be used to build reusable pieces of code. Let´s know take a look at: - inheritance - modules - import

1 Inheritance

One of the most important goals of the object-oriented approach to programming is the creation of **stable, reliable, reusable code**. If you had to create a new class for every kind of object you wanted to model, you would hardly have any reusable code. In Python and any other language that supports OOP, one class can **inherit** from another class. This means you can base a new class on an existing class; the new class *inherits* all of the attributes and behavior of the class it is based on. A new class can override any undesirable attributes or behavior of the class it inherits from, and it can **add any new attributes or behavior** that are appropriate. The original class is called the **parent** class, and the new class is a **child** of the parent class. The parent class is also called a **superclass**, and the child class is also called a **subclass**.

The child class inherits all attributes and behavior from the parent class, but **any attributes that are defined *only* in the child class** are not available to the parent class. This may be obvious to many people, but it is worth stating. This also means **a child class can override behavior of the parent class**. If a child class defines a method that also appears in the parent class, objects of the child class will use the new method rather than the parent class method.

To better understand inheritance, let's look at an example of a class that can be based on the Rocket class.

1.1 The SpaceShuttle class

If you wanted to model a space shuttle, you could write an entirely new class. But **a space shuttle is just a special kind of rocket.** Instead of writing an entirely new class, you can inherit all of the attributes and behavior of a Rocket, and then add a few appropriate attributes and behavior for a Shuttle.

One of the most significant characteristics of a space shuttle is that it can be reused. So the only difference we will add at this point is to **record the number of flights the shutttle has completed**. Everything else you need to know about a shuttle has already been coded into the Rocket class.

Here is what the Shuttle class looks like:

[1]:
```
###highlight=[25,26,27,28,29,30,31,32,33,34]
from math import sqrt

class Rocket():
    # Rocket simulates a rocket ship for a game,
    #  or a physics simulation.

    def __init__(self, x=0, y=0):
        # Each rocket has an (x,y) position.
        self.x = x
        self.y = y

    def move_rocket(self, x_increment=0, y_increment=1):
        # Move the rocket according to the paremeters given.
        #  Default behavior is to move the rocket up one unit.
        self.x += x_increment
        self.y += y_increment

    def get_distance(self, other_rocket):
        # Calculates the distance from this rocket to another rocket,
        #  and returns that value.
        distance = sqrt((self.x-other_rocket.x)**2+(self.y-other_rocket.y)**2)
        return distance

# The new class Shuttle is inherete from class Rocket
class Shuttle(Rocket):
    # Shuttle simulates a space shuttle, which is really
    #  just a reusable rocket.

    def __init__(self, x=0, y=0, flights_completed=0):
        super().__init__(x, y)
        self.flights_completed = flights_completed

# invoking class Shuttle to create object shuttle
shuttle = Shuttle(10,0,3)
print(shuttle)
```

```
<__main__.Shuttle object at 0x7f62940e3fa0>
```

When a new class is based on an existing class, you write the name of the parent class in parentheses when you define the new class:

class NewClass(ParentClass):

The ___init___() function of the new class needs to call the ___init___() function of the parent class. The ___init___() function of the new class needs to accept all of the parameters required to build an object from the parent class, and these parameters need to be passed to the ___init___() function of the parent class. The **super().___init___()** function takes care of this:

[2]:
```
# this is just to introduce the syntax of inherited classes
# it will generate an error if you run it on it´s own
# try uncommenting to see the type of error generated

#class NewClass(ParentClass):
#
#    def __init__(self, arguments_new_class, arguments_parent_class):
#        super().__init__(arguments_parent_class)
#        # Code for initializing an object of the new class.
```

The *super()* function passes the *self* argument to the parent class automatically. You could also do this by explicitly naming the parent class when you call the ___init___() function, but you then have to include the *self* argument manually:

[3]:
```
###highlight=[7]
class Shuttle(Rocket):
    # Shuttle simulates a space shuttle, which is really
    #  just a reusable rocket.

    def __init__(self, x=0, y=0, flights_completed=0):
        Rocket.__init__(self, x, y)
        self.flights_completed = flights_completed
```

This might seem a little easier to read, but it is preferable to use the *super()* syntax. **When you use *super()*, you don't need to explicitly name the parent class**, so your code is more resilient to later changes. As you learn more about classes, you will be able to write child classes that inherit from multiple parent classes, and the *super()* function will call the parent classes' ___init___() functions for you, in one line. This explicit approach to calling the parent class' ___init___() function is included so that you will be less confused if you see it in someone else's code.

The output above shows that a new Shuttle object was created. This new Shuttle object can store the number of flights completed, but it also has all of the functionality of the Rocket class: it has a position that can be changed, and it can calculate the distance between itself and other rockets or shuttles. This can be demonstrated by creating several rockets and shuttles, and then finding the distance between one shuttle and all the other shuttles and rockets. This example uses a simple function called randint, which generates a random integer between a lower and upper bound, to determine the position of each rocket and shuttle:

[4]:
```
from math import sqrt

# randint generates a random integer between a lower and upper bound
```

```
from random import randint

class Rocket():
    # Rocket simulates a rocket ship for a game,
    #  or a physics simulation.

    def __init__(self, x=0, y=0):
        # Each rocket has an (x,y) position.
        self.x = x
        self.y = y

    def move_rocket(self, x_increment=0, y_increment=1):
        # Move the rocket according to the paremeters given.
        #  Default behavior is to move the rocket up one unit.
        self.x += x_increment
        self.y += y_increment

    def get_distance(self, other_rocket):
        # Calculates the distance from this rocket to another rocket,
        # and returns that value.
        distance = sqrt((self.x-other_rocket.x)**2+(self.y-other_rocket.y)**2)
        return distance

class Shuttle(Rocket):
    # Shuttle simulates a space shuttle, which is really
    # just a reusable rocket.

    def __init__(self, x=0, y=0, flights_completed=0):
        super().__init__(x, y)
        self.flights_completed = flights_completed


# Create several shuttles and rockets, with random positions.
# Shuttles have a random number of flights completed.
shuttles = []
for x in range(0,3):
    x = randint(0,100)
    y = randint(1,100)
    flights_completed = randint(0,10)
    shuttles.append(Shuttle(x, y, flights_completed))

rockets = []
for x in range(0,3):
    x = randint(0,100)
    y = randint(1,100)
    rockets.append(Rocket(x, y))
```

```
# Show the number of flights completed for each shuttle.
for index, shuttle in enumerate(shuttles):
    print("Shuttle %d has completed %d flights." % (index, shuttle.
 ↪flights_completed))

print("\n")
# Show the distance from the first shuttle to all other shuttles.
first_shuttle = shuttles[0]
for index, shuttle in enumerate(shuttles):
    distance = first_shuttle.get_distance(shuttle)
    print("The first shuttle is %f units away from shuttle %d." % (distance,␣
 ↪index))

print("\n")
# Show the distance from the first shuttle to all other rockets.
for index, rocket in enumerate(rockets):
    distance = first_shuttle.get_distance(rocket)
    print("The first shuttle is %f units away from rocket %d." % (distance,␣
 ↪index))
```

```
Shuttle 0 has completed 0 flights.
Shuttle 1 has completed 3 flights.
Shuttle 2 has completed 1 flights.


The first shuttle is 0.000000 units away from shuttle 0.
The first shuttle is 65.764732 units away from shuttle 1.
The first shuttle is 49.010203 units away from shuttle 2.


The first shuttle is 42.720019 units away from rocket 0.
The first shuttle is 54.120237 units away from rocket 1.
The first shuttle is 76.557168 units away from rocket 2.
```

Inheritance is a powerful feature of object-oriented programming. Using just what you have seen so far about classes, you can model an incredible variety of real-world and virtual phenomena with a high degree of accuracy. The code you write has the potential to be stable and reusable in a variety of applications.

1.2 Inheritance in Python 2.7

The *super()* method has a slightly different syntax in Python 2.7. You will be using Python 3, but it may be useful to recognize a different syntax, in case you bump into older code snippets.

[5]:
```
###highlight=[5]
# this is exemplary and will generate an error if you run it

#class NewClass(ParentClass):
```

```
#
#    def __init__(self, arguments_new_class, arguments_parent_class):
#        super(NewClass, self).__init__(arguments_parent_class)
#        # Code for initializing an object of the new class.
```

Notice that you have to explicitly pass the arguments *NewClass* and *self* when you call *super()* in Python 2.7. The SpaceShuttle class would look like this:

[2]:
```
###highlight=[25,26,27,28,29,30,31,32,33,34]
from math import sqrt

class Rocket(object):
    # Rocket simulates a rocket ship for a game,
    #  or a physics simulation.

    def __init__(self, x=0, y=0):
        # Each rocket has an (x,y) position.
        self.x = x
        self.y = y

    def move_rocket(self, x_increment=0, y_increment=1):
        # Move the rocket according to the paremeters given.
        #  Default behavior is to move the rocket up one unit.
        self.x += x_increment
        self.y += y_increment

    def get_distance(self, other_rocket):
        # Calculates the distance from this rocket to another rocket,
        #  and returns that value.
        distance = sqrt((self.x-other_rocket.x)**2+(self.y-other_rocket.y)**2)
        return distance

class Shuttle(Rocket):
    # Shuttle simulates a space shuttle, which is really
    #  just a reusable rocket.

    def __init__(self, x=0, y=0, flights_completed=0):
        super(Shuttle, self).__init__(x, y)
        self.flights_completed = flights_completed

shuttle = Shuttle(10,0,3)
print(shuttle)
```

```
<__main__.Shuttle object at 0x7f62940c9640>
```

As you can see, **this syntax works in Python 3** as well.

1.3 # Exercise

Student Class

- Start with your program from the Person Class created in an earlier lecture.
- Make a new class called Student that inherits from Person.
- Define some attributes that a student has, which other people don't have.
  - A student has a school they are associated with, a graduation year, a gpa, and other particular attributes.
- Create a Student object, and prove that you have used inheritance correctly.
  - Set some attribute values for the student, that are only coded in the Person class.
  - Set some attribute values for the student, that are only coded in the Student class.
  - Print the values for all of these attributes.

[3]:
```
# write your code below
# DON´T DELETE THIS LINE PLEASE ****
#
class person():

    def __init__(self,name ="mohsen",age =25,birth_country = "iran"):
        self.name = name
        self.age = age
        self.birth_country = birth_country

    def introduce(self):
        self.name = "mohsen"
        self.age = 25
        print(f"Hello my name is {self.name} and i have {self.age} years old")
    def age_person(self):
        self.age += 1

class student(person):

    def␣
 ↪__init__(self,name="mohsen",age=25,birth_country="iran",university_name="university␣
 ↪of bolzano",graduation_year="2023"):
        super().__init__(name,age,birth_country)
        self.university_name=university_name
        self.graduation_year=graduation_year

student1 = student("mohsen",25,"iran","university of bolzano","2023")
print(student1.name,student1.age,student1.birth_country,student1.
 ↪university_name,student1.graduation_year)
```

```
mohsen 25 iran university of bolzano 2023
```

1.4 # Exercise

Refining Shuttle

- Take the latest version of the Shuttle class. Extend it as follows:
  - Add more attributes that are particular to shuttles such as maximum number of flights, capability of supporting spacewalks, and capability of docking with the ISS.
  - Add one more method to the class, that relates to shuttle behavior. This method could simply print a statement, such as "Docking with the ISS," for a dock_ISS() method.
  - Prove that your refinements work by creating a Shuttle object with these attributes, and then call your new method.

[4]:
```
# write your code below
# DON´T DELETE THIS LINE PLEASE ****
#

class Shuttle(Rocket):

    def __init__(self, x=0, y=0,␣
 ↪flights_completed=0,max_flights=3,spacewalk_support=True,issdock_support=True):
 ↪
        super().__init__(x, y)
        self.flights_completed = flights_completed
        self.max_flights = max_flights
        self.spacewalk_support = spacewalk_support
        self.issdock_support = issdock_support
    def dock_iss(self):
        print("Docking with the ISS")
shuttle1 = Shuttle(5,4,5,7,True,True)
print(shuttle1.x,shuttle1.y,shuttle1.flights_completed,shuttle1.
 ↪max_flights,shuttle1.spacewalk_support,shuttle1.issdock_support)
shuttle1.dock_iss()
```

```
5 4 5 7 True True
Docking with the ISS
```

2 Modules and classes

Now that you are starting to work with classes, your files are going to grow longer. This is good, because it means your programs are probably doing more interesting things. But it is bad, because longer files can be more difficult to work with. Python allows you to save your classes in another file and then import them into the program you are working on. This has the added advantage of isolating your classes into files that can be used in any number of different programs. As you use your classes repeatedly, the classes become more reliable and complete overall.

2.1 Storing a single class in a module

When you save a class into a separate file, that file is called a **module**. You can have any number of classes in a single module. There are a number of ways you can then import the class you are interested in.

Start out by saving just the Rocket class into a file called *rocket.py*. **Notice the naming con-**

vention being used here: the *module* is saved with a *lowercase name*, and the *class* starts with an *uppercase letter*. This convention is pretty important for a number of reasons, and it is a really good idea to follow the convention.

[5]:
```
###highlight=[2]
# Save as rocket.py
# this creates a new module rocket where the class Rocket is defined

from math import sqrt

class Rocket():
    # Rocket simulates a rocket ship for a game,
    #  or a physics simulation.

    def __init__(self, x=0, y=0):
        # Each rocket has an (x,y) position.
        self.x = x
        self.y = y

    def move_rocket(self, x_increment=0, y_increment=1):
        # Move the rocket according to the paremeters given.
        #  Default behavior is to move the rocket up one unit.
        self.x += x_increment
        self.y += y_increment

    def get_distance(self, other_rocket):
        # Calculates the distance from this rocket to another rocket,
        #  and returns that value.
        distance = sqrt((self.x-other_rocket.x)**2+(self.y-other_rocket.y)**2)
        return distance
```

2.2 # Exercise

How do you actully create this new module in JHUB?

1. create a file of type text **within the same directory** of this present notebook
2. name the new file as: rocket.py (this is a Python code not a notebook)
3. cut and paste the class as defined in the previous code cell
4. to test out that you have done things correctly, try importing the class Rocket from the module rocket. The following code cell is doing exactly that. If you get an error, then you need to find out the problem.

Make a separate file (a new **module**) called *rocket_game.py*. Again, to use standard naming conventions, make sure you are using a lowercase_underscore name for this file.

[6]:
```
# Here we are creating a new module: rocket_game
# which is importing the class rocket and instantiating an object and printing␣
 ↪its location
# Save as rocket_game.py
```

```
from rocket import Rocket

rocket = Rocket()
print("The rocket is at (%d, %d)." % (rocket.x, rocket.y))
```

```
The rocket is at (0, 0).
```

This is a really clean and uncluttered file. A rocket is now something you can define in your programs, without the details of the rocket's implementation cluttering up your file. You don't have to include all the class code for a rocket in each of your files that deals with rockets; the code defining rocket attributes and behavior lives in one file, and can be used anywhere.

The first line tells Python to look for a file called *rocket.py*. It looks for that file in the same directory as your current program. You can put your classes in other directories, but we will get to that convention a bit later.

When Python finds the file *rocket.py*, it looks for a class called *Rocket*. When it finds that class, it imports that code into the current file, without you ever seeing that code. You are then free to use the class Rocket as you have seen it used in previous examples.

2.3 # Exercise

Test out your new modules

1. Clear the whole memory (KERNEL – Restart & clear output)
2. Run the following cell.
3. if you get an error, you need to find out how to fix it

[7]:
```
# This is running a python script directly into the Notebook
%run rocket_game.py
```

```
The rocket is at (0, 0).

The shuttle is at (0, 0).
The shuttle has completed 0 flights.
```

DON´T DELETE THIS LINE PLEASE

WRITE DOWN THE EXACT OUTPUT OBTAINED WHEN YOU RUN THE ABOVE **code cell**

2.4 Storing multiple classes in a module

A module is simply a file that contains one or more classes or functions, so the Shuttle class actually belongs in the rocket module as well:

[9]:
```
###highlight=[27,28,29,30,31,32,33]
# Save as rocket.py
from math import sqrt
```

```
class Rocket():
    # Rocket simulates a rocket ship for a game,
    #  or a physics simulation.

    def __init__(self, x=0, y=0):
        # Each rocket has an (x,y) position.
        self.x = x
        self.y = y

    def move_rocket(self, x_increment=0, y_increment=1):
        # Move the rocket according to the paremeters given.
        #  Default behavior is to move the rocket up one unit.
        self.x += x_increment
        self.y += y_increment

    def get_distance(self, other_rocket):
        # Calculates the distance from this rocket to another rocket,
        #  and returns that value.
        distance = sqrt((self.x-other_rocket.x)**2+(self.y-other_rocket.y)**2)
        return distance


class Shuttle(Rocket):
    # Shuttle simulates a space shuttle, which is really
    #  just a reusable rocket.

    def __init__(self, x=0, y=0, flights_completed=0):
        super().__init__(x, y)
        self.flights_completed = flights_completed
```

Now you can import the Rocket and the Shuttle class, and use them both in a clean uncluttered program file:

[8]:
```
###highlight=[3,8,9,10]
# Save as rocket_game.py
from rocket import Rocket, Shuttle

rocket = Rocket()
print("The rocket is at (%d, %d)." % (rocket.x, rocket.y))

shuttle = Shuttle()
print("\nThe shuttle is at (%d, %d)." % (shuttle.x, shuttle.y))
print("The shuttle has completed %d flights." % shuttle.flights_completed)
```

```
The rocket is at (0, 0).

The shuttle is at (0, 0).
The shuttle has completed 0 flights.
```

The first line tells Python to import both the *Rocket* and the *Shuttle* classes from the *rocket* module. You don't have to import every class in a module; you can pick and choose the classes you care to use, and Python will only spend time processing those particular classes.

[13]:
```
# This is running a python script directly into the Notebook
%run rocket_game.py
```

```
The rocket is at (0, 0).

The shuttle is at (0, 0).
The shuttle has completed 0 flights.
```

2.5 A number of ways to import modules and classes

There are several ways to import modules and classes, and each has its own merits.

2.5.1 import *module_name*

The syntax for importing classes that was just shown:

from module_name import ClassName

for example:

[10]:
```
# importing just one of the two classes
from rocket import Rocket
```

[11]:
```
# importing both classes in one go
from rocket import Rocket, Shuttle
```

This is straightforward, and is used quite commonly. It allows you to use the class names directly in your program, so you have very clean and readable code.

This can be a **problem if the names of the classes you are importing conflict with names that have already been used in the program you are working on**.

This is unlikely to happen in the short programs you have been seeing here, but if you were working on a larger program it is quite possible that the class you want to import from someone else's work would happen to have a name you have already used in your program. In this case, you can simply import the module itself:

[12]:
```
# Save as rocket_game.py
import rocket

rocket_0 = rocket.Rocket()
print("The rocket is at (%d, %d)." % (rocket_0.x, rocket_0.y))

shuttle_0 = rocket.Shuttle()
print("\nThe shuttle is at (%d, %d)." % (shuttle_0.x, shuttle_0.y))
print("The shuttle has completed %d flights." % shuttle_0.flights_completed)
```

```
The rocket is at (0, 0).

The shuttle is at (0, 0).
The shuttle has completed 0 flights.
```

The general syntax for this kind of import is:

import module_name

After this, classes are accessed using dot notation:

module_name.ClassName

This prevents some name conflicts.

If you were reading carefully however, you might have noticed that **the variable name *rocket* in the previous example had to be changed** because it has the same name as the module itself. This is not good, because in a longer program that could mean a lot of renaming.

2.5.2 import *module_name* as *local_module_name*

There is another syntax for imports that is quite useful:

import module_name as local_module_name

When you are importing a module into one of your projects, you are free to choose any name you want for the module in your project. So the last example could be rewritten in a way that the variable name *rocket* would not need to be changed:

[13]:
```
# Save as rocket_game.py
import rocket as rocket_module

rocket = rocket_module.Rocket()
print("The rocket is at (%d, %d)." % (rocket.x, rocket.y))

shuttle = rocket_module.Shuttle()
print("\nThe shuttle is at (%d, %d)." % (shuttle.x, shuttle.y))
print("The shuttle has completed %d flights." % shuttle.flights_completed)
```

```
The rocket is at (0, 0).

The shuttle is at (0, 0).
The shuttle has completed 0 flights.
```

This approach is often used to shorten the name of the module, so you don't have to type a long module name before each class name that you want to use. But it is easy to shorten a name so much that you force people reading your code to scroll to the top of your file and see what the shortened name stands for. In this example,

[14]:
```
import rocket as rocket_module
```

leads to much more readable code than something like:

[15]:
```
import rocket as r
```

2.5.3 from *module_name* import *

There is one more import syntax that you should be aware of, but you should probably avoid using. This syntax imports all of the available classes and functions in a module:

[21]:
```
# from module_name import *
```

This is not recommended, for a couple reasons. First of all, you may have no idea what all the names of the classes and functions in a module are. If you accidentally give one of your variables the same name as a name from the module, you will have **naming conflicts**. Also, you may be **importing way more code into your program than you need**.

If you really need all the functions and classes from a module, just import the module and use the `module_name.ClassName` syntax in your program.

You will get a sense of how to write your imports as you read more Python code, and as you write and share some of your own code.

2.6 A module of functions

You can use modules to store a set of functions you want available in different programs as well, even if those functions are not attached to any one class. To do this, you save the functions into a file, and then import that file just as you saw in the last section. Here is a really simple example; save this is *multiplying.py*:

[16]:
```
# Save as multiplying.py
def double(x):
    return 2*x

def triple(x):
    return 3*x

def quadruple(x):
    return 4*x
```

Now you can import the file *multiplying.py*, and use these functions. Using the `from module_name import function_name` syntax:

[17]:
```
###highlight=[2]
from multiplying import double, triple, quadruple

print(double(5))
print(triple(5))
print(quadruple(5))
```

```
10
15
20
```

Using the `import module_name` syntax:

[18]:
```
###highlight=[2]
import multiplying

print(multiplying.double(5))
print(multiplying.triple(5))
print(multiplying.quadruple(5))
```

```
10
15
20
```

Using the `import module_name as local_module_name` syntax:

[25]:
```
###highlight=[2]
import multiplying as m

print(m.double(5))
print(m.triple(5))
print(m.quadruple(5))
```

```
10
15
20
```

Using the `from module_name import *` syntax:

[26]:
```
###highlight=[2]
from multiplying import *

print(double(5))
print(triple(5))
print(quadruple(5))
```

```
10
15
20
```

2.7 Exercise

Importing Student

- Take your program from the Student Class (used in a previous Notebook):
  - Save your Person and Student classes in a separate file called *person.py*.
  - Save the code that uses these classes in four separate files.
    - In the first file, use the `from module_name import ClassName` syntax to make your program run.
    - In the second file, use the `import module_name` syntax.

    * In the third file, use the `import module_name as different_local_module_name` syntax.
    * In the fourth file, use the `import *` syntax.

2.8 Exercise

Importing Car

- Take your program from the Car Class (used in a previous Notebook):
  - Save your Car class in a separate file called *car.py*.
  - Save the code that uses the car class into four separate files.
    * In the first file, use the `from module_name import ClassName` syntax to make your program run.
    * In the second file, use the `import module_name` syntax.
    * In the third file, use the `import module_name as different_local_module_name` syntax.
    * In the fourth file, use the `import *` syntax.

3 Revisiting PEP 8

If you recall, PEP 8 is the style guide for writing Python code. PEP 8 has a little to say about writing classes and using `import` statements, that was not covered previously. Following these guidelines will help make your code readable to other Python programmers, and it will help you make more sense of the Python code you read.

3.1 Import statements

PEP8 provides clear guidelines about where import statements should appear in a file. The names of modules should be on separate lines:

[19]:
```
# this
import sys
import os

# not this
import sys, os
```

The names of classes can be on the same line:

[28]:
```
from rocket import Rocket, Shuttle
```

Imports should always be **placed at the top of the file**. When you are working on a longer program, you might have an idea that requires an import statement. You might write the import statement in the code block you are working on to see if your idea works. If you end up keeping the import, make sure you move the import statement to the top of the file. This lets anyone who works with your program see what modules are required for the program to work.

Your import statements should be in a predictable order:

- The first imports should be standard Python modules such as *sys*, *os*, and *math*.

- The second set of imports should be "third-party" libraries. These are libraries that are written and maintained by independent programmers, which are not part of the official Python language. An example of this is pygame.

3.2 Module and class names

Modules should have short, lowercase names. If you want to have a space in the module name, use an underscore.

Class names should be written in *CamelCase*, with an initial capital letter and any new word capitalized. There should be no underscores in your class names.

This convention **helps distinguish modules from classes**, for example when you are writing import statements.

3.3 Exercise

PEP 8 Compliance

- Take a look at each of the programs you have written for this section, and make sure they comply with the guidelines from PEP 8.
  - Make sure your import statements are formatted properly, and appear at the top of the file.
  - Make sure your class names are formatted properly.
  - Make sure your module names are formatted properly.