Writing the Program

by Sophia

WHAT'S COVERED

In this lesson, we will learn about the completed program for the application that we've defined. Specifically, this lesson covers:

1. Writing Core Functionality

BIG IDEA

The creation of the entire program from start to finish should ideally be done in parts. Try to focus on writing the code around a single comment before moving on. It will help make that process easier. Each program will be different. What we'll do here is explore the demonstration program and talk through the code for it, so you can apply some of those same steps to your program.

In the last lesson, we created a good solid framework of the application and used this to build out some of the functionality of the main program. We completed each of those steps and now have a larger program. We now want to break our code up even further.

Although a program may work as it already is, we may want to modularize our code for easier future reuse. Certain pieces may be easy to reuse as modules in other programs. For example, if we wanted to use the functionality for a die outside of casino craps, we could easily do so. This gives us a lot more flexibility to do so by keeping all of the elements of the code consistent between implementations. What we will end up doing is splitting our code into individual reusable modules including a `die` module. The `die` module will contain a `Die` class that consists of everything about the die. We'll also create a `player` module that will consist of a `Player` class. The `Player` class consists of the code about the player itself and the functionality in the play.

THINK ABOUT IT

How you break up the modules is up to you but typically the functionality of a module is individualized and set up specifically for that class. There are many reasons why this is done including the manageability of the code. When we create a larger problem, it can be difficult to stay focused on a single piece of code. When we break things down into individual tasks, it becomes less overwhelming as we start to develop the code. Also by breaking things down, we have the ability to work with others on a larger problem. Since the work from different people can be compiled together to create a larger program, we can develop things a lot faster. The reuse of the modules ensures that we can reuse parts that already work. If we have a piece of code that works well for a particular function, we don't have to redevelop that over and over again.

For our demonstration program, we will continue this lesson by creating two modules that we can import from, a file to save to, and the main program that runs the game. So, we will be moving from file to file as we continue. First up is the `die` module.

2. die Module

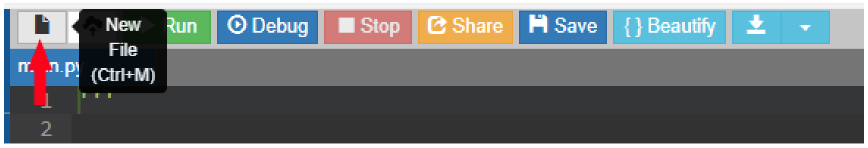
One key class that we can define is the `Die` class. The face value of the die should set itself with the initial value of 1 (since a die does not have a 0 value on it) and have a roll method that returns an integer value between 1 and 6. Remember, to get this random number between 1 and 6, we will need to import the `random` module or at least the `randint` function of the `random` module.

Before we add any code, we would like this `Die` class to be its own module so we can import it into other programs if we desire. To do that we need to remember back to Unit 3's Introduction to I/O lesson. Since this class (to be a module) needs to "sit" outside of the main.py file, we need to add a file.

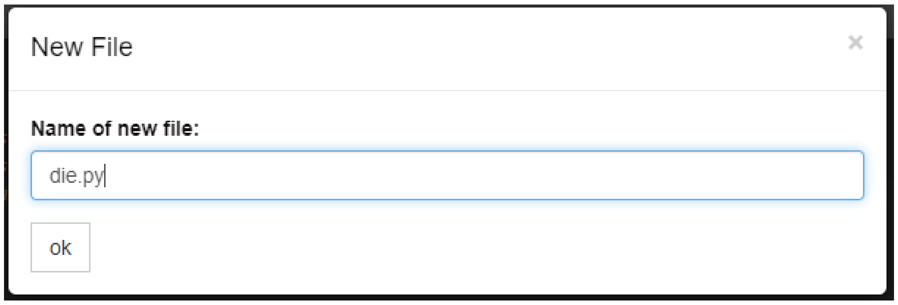
TRY IT

Directions: If you recall, we first need to add a file to make this a module that we can reuse by importing from. Follow the steps below to create the `die` module.

1. Click on the 'Add file' icon at the top of the Files panel.



2. Next, we name the file. In this case, we will call it die.


New File

Name of new file:

die.py

ok

Note: Ensure that the filename ends with ".py".

Now that we have the .py file (module) created, let's see what this class would look like inside:

EXAMPLE

```
#Importing the randint module
from random import randint

#The class Die implements a six-sided die

class Die:

    #The __init__ method is used to create a new instance of die with a default value of 1
    def __init__(self):
        self.value = 1

    #The roll method is used to set the value to a random number between 1 and 6
    def roll(self):
        self.value = randint(1, 6)

    #The getValue method returns the top face value of the die
    def getValue(self):
        return self.value

    #The __str__ method returns the string representation of the value of the die
    def __str__(self):
        return str(self.getValue())
```

TRY IT

Directions: Go ahead and add the `Die` class to the die.py file.

This is the foundation of the `Die` class with all of the methods that would be used. We've also added the `.getValue()` method that returns the value that has been rolled. At this point we are done with this `die` module. Let's move over to the `player` module.

3. player Module

We want to be able to reuse this module as well, so we will need to follow the same steps as we did in the `die` module section and create a file called *player.py*.

TRY IT

Directions: Go ahead and add a file called player.py.

Next, we'll create a class called `Player` in that `player` module. This class is going to simulate what a player can do. Because the die functionality is located in the `Die` class of the `die` module, we'll have to import the `Die` class from that module before we create the `Player` class:

EXAMPLE

```
#The player class is to simulate what a player is able to do.

#We will first import the class Die that we have created
from die import Die

#The class Player is the functionality that makes use of the Die class.
class Player(object):
```

TRY IT

Directions: Enter the code in the `player` module to import the `Die` and start the `Player` class.

Next, we'll define the `__init__` method which will create and initialize the variables that we'll need for this class. One key thing to note is that we're defining two separate `Die` variables to play the game correctly since casino craps use two dice.

```
    #The __init__ method will create a pair of dice and initialize the other variables.
    def __init__(self):
        self.firstDie = Die()
        self.secondDie = Die()
        self.roll = ""
        self.rollsCount = 0
        self.startOfGame = True
        self.winner = self.loser = False
```

Generally, for any variables that we need to access, we'll use a method that returns the value. In this case, the number of rolls will be returned with the `.getNumberOfRolls()` method we will define and code.

```
    #The getNumberOfRolls will return the rollCount for the number of rolls
    def getNumberOfRolls(self):
        return self.rollsCount
```

Next, we'll implement one of the core items of the program for the game itself. Remember that one of the most important parts of writing a program is to ensure that the comments explain the code:

```
    #The rollDice function rolls both of the dice once.
    #Then it updates the roll, the won and the lost outcomes.
    #Lastly it returns a tuple of the values of the dice.
    def rollDice(self):
        #Increment rollCount by 1
        self.rollsCount += 1

        #Roll both dice
        self.firstDie.roll()
        self.secondDie.roll()

        #Set the tuple based on the values from each dice
        (v1, v2) = (self.firstDie.getValue(),
                     self.secondDie.getValue())

        #Set the roll value to the value of the two dice
        self.roll = str((v1, v2)) + " total = " + str(v1 + v2)

        #The logic of the game is now running, if this is the first game
        if self.startOfGame:
            #Initial value is set to the the value of the two dice
            self.initialSum = v1 + v2
            self.startOfGame = False

            #If the initial sum is equal to 2, 3 or 12, the player is set to have lost
            if self.initialSum in (2, 3, 12):
                self.loser = True
            #If the initial sum is equal to 7 or 11, the player is set to have won
            elif self.initialSum in (7, 11):
                self.winner = True
        #If this is not the first game
        else:
            #We are now checking for the later sum of the values
            laterSum = v1 + v2
            #If it's not the start of the game and a 7 is rolled, the player loses
            if laterSum == 7:
                self.loser = True
            #If the player rolled the same sum as the first sum, the player wins
            elif laterSum == self.initialSum:
                self.winner = True
        return (v1, v2)
```

Note: This function returns a tuple versus a list because of the way that data is returned from a function. If you remember from Unit 2, tuples are the same as a list; however, they are unchangeable (unmutable). Once we rolled our dice, we wouldn't want another function to change the values of those die values, thus a tuple as the data collection type works well here.

Using the original logic of the program, most of this should make sense. The only key difference here is that we are storing both rolls in a tuple so that we can track each die separately.

We’ll also check if the player has won or lost.

```
   def isWinner(self):
        """Returns True if player has won."""
        return self.winner

    def isLoser(self):
        """Returns True if player has lost."""
        return self.loser
```

Note: The three double quotes, if you remember back to Unit 3’s Creating Modules lesson, refer to a Python docstring. This is used to document a Python module, class function, or method. This is used so that programmers can understand what it does without having to read all of the implementation. In that Creating Modules lesson, we also learned that you can use the `help()` function to find out the details about the documentation within the docstring.

Note that it's possible after one roll that both `isWinner()` and `isLoser()` are false. If this is the case, the game is not finished and we must roll again until one is true. The following method will handle that for us:

```
    #Plays a full game and counts the rolls for that game.
    #This returns True for a win and False if the player loses
    def play(self):
        #We continue to roll as long as both are False
        while not self.isWinner() and not self.isLoser():
            self.rollDice()
        return self.isWinner()
```

Lastly, we have a `__str__` method to return a string value of the last roll. The **__str__** method is used for classes and returns back a string value that the developer defines. In our case, we are utilizing it to return the value of the die as a string, instead of variable.roll outputting the variable the `__str__` method returns. We could just convert it when we need it, but using this approach helps to simplify the results.

```
    #The __str__ method will return the last roll as a string.
    def __str__(self):
        """Returns a string representation of the last roll."""
        return self.roll
```

TRY IT

Directions: Enter the rest of the code for the `Player` class. The complete code is shown below.

```
#The player class is to simulate what a player is able to do.

#We will first import the class Die that we have created
from die import Die

#The class Player is the functionality that makes use of the Die class.
class Player(object):

    #The __init__ method will create a pair of dice and initialize the other variables.
    def __init__(self):
        self.firstDie = Die()
        self.secondDie = Die()
        self.roll = ""
        self.rollsCount = 0
        self.startOfGame = True
        self.winner = self.loser = False

    #The getNumberOfRolls will return the rollCount for the number of rolls
    def getNumberOfRolls(self):
        return self.rollsCount
    #The rollDice rolls both of the dice once.
    #Then it updates the roll, the won and the lost outcomes.
    #Lastly it returns a tuple of the values of the dice.
    def rollDice(self):
        #Increment rollCount by 1
        self.rollsCount += 1

        #Roll both dice
        self.firstDie.roll()
        self.secondDie.roll()

        #Set the tuple based on the values from each dice
        (v1, v2) = (self.firstDie.getValue(),
                    self.secondDie.getValue())

        #Set the roll value to the value of the two dice
        self.roll = str((v1, v2)) + " total = " + str(v1 + v2)

        #The logic of the game is now running, if this is the first game
        if self.startOfGame:
            #Initial value is set to the the value of the two dice
            self.initialSum = v1 + v2
            self.startOfGame = False

            #If the initial sum is equal to 2, 3 or 12, the player is set to have lost
            if self.initialSum in (2, 3, 12):
                self.loser = True
            #If the initial sum is equal to 7 or 11, the player is set to have won
            elif self.initialSum in (7, 11):
                self.winner = True
        #If this is not the first game
        else:
            #We are now checking for the later sum of the values
            laterSum = v1 + v2
            #If it's not the start of the game and a 7 is rolled, the player loses
            if laterSum == 7:
                self.loser = True
            #If the player rolled the same sum as the first sum, the player wins
            elif laterSum == self.initialSum:
                self.winner = True
        return (v1, v2)

    def isWinner(self):
        """Returns True if player has won."""
        return self.winner
    def isLoser(self):
        """Returns True if player has lost."""
        return self.loser

    #With the game, it's possible that both isWinner and isLower is false
    #If this is the case, the game is not finished and we must roll again until one is true.

    #Plays a full game and counts the rolls for that game.
    #This returns True for a win and False if the player loses
    def play(self):
        #We continue to roll as long as both are False
        while not self.isWinner() and not self.isLoser():
            self.rollDice()
        return self.isWinner()

    #The __str__ method will return the last roll as a string.
    def __str__(self):
        """Returns a string representation of the last roll."""
        return self.roll
```

TERM TO KNOW

`__str__`
The `__str__` method is used for classes and returns back a string value that the developer defines.

4. Finishing the Main Program

Lastly, we'll finish up the coding in our main program. Back in the main.py file, we'll see some new things here as part of the program.

```
#This game plays the game of dice called craps where
#Players would bet on the outcomes of a pair of dice rolls.
#If the sum of the dice is 2, 3 or 12, the player loses immediately.
#If the sum of the dice is 7 or 11, they win immediately.
#The purpose of the program is to simulate the results between two players

from player import Player

#Importing the datetime to get the current date and time
from datetime import datetime
```

Here we are importing the `Player` class from the `player` module. We are also importing the `datetime()` function from the `datetime` module so we can identify when the game was run.

Inside the "main" program (the main.py file), we're going to explain what the code is meant to be doing with comments. This way, any time that anyone else reviews the code, it will be obvious what the code is doing. Then we'll create a function that's meant to play a single game. This way, we can fully test it. Here is the code for single gameplay.

```
#This function is created to play a single game and print out the results after each roll.
def playOneGame():

    player = Player()
    while not player.isWinner() and not player.isLoser():
        player.rollDice()
        print(player)
    if player.isWinner():
        print("You win!")
    else:
        print("You lose!")
```

Next, we'll create a new function that plays multiple games. This is the piece of code that we should recognize from the last lesson, but it's been defined a bit better now:

```
#This function is created to play multiple games and outputs the results
#based on the number of games selected.
def playMultipleGames(number):
    #Initializing the variables
    wins = 0
    losses = 0
    winRolls = 0
    lossRolls = 0

    #Looping through the number of executions
    for count in range(number):
        player = Player()
        hasWon = player.play()
        rolls = player.getNumberOfRolls()
        if hasWon:
            wins += 1
            winRolls += rolls
        else:
            losses += 1
            lossRolls += rolls

    #Calculating the statistics
    print("The total number of wins is", wins)
    print("The total number of losses is", losses)
    print("The average number of rolls per win is %0.2f" % \
          (winRolls / wins))
    print("The average number of rolls per loss is %0.2f" % \
          (lossRolls / losses))
    print("The winning percentage is %0.3f" % (wins / number))
    print("The multi-game has been saved into the log.")
    logStats(wins, losses, winRolls, lossRolls, number)
```

Note: If you recall back to Unit 2's Debugging Function lesson, the `2f` provided a double decimal.

The last function we will call `logStats()`. The purpose of this function is to store the results to a file. Remember this was a requirement for this program since our friend wanted the statistics saved to a file.

THINK ABOUT IT

If you remember from Unit 3's Reading and Writing to a File lesson, we identified that when using the open() function there are a few different modes that can be set when opening the files. These included the 'r' (reading from a file), 'w' (writing to a file), and the 'a' (appending to a file). Both writing and appending modes create the file if it does not yet exist. However, using the write mode, if the file does exist, the file is cleared out completely. That might be a good mode to use if you are looking for a log that only contains the most recent statistics output. For our program, we want to log all the runs, so we'll use the append mode that will not clear the file but rather continue to add (append) to it.

Let's create this new `logStats()` function with the append mode:

```
#This function will log each multi game run to include the date/time and the details of the run
def logStats(wins, losses, winRolls, lossRolls, number):
    file = open("log.txt", "a")

    #Create a date/time object to get the current date and time
    now = datetime.now()
    #Formatting the output of the date and time to dd/mm/YY H:M:S
    dt_string = now.strftime("%d/%m/%Y %H:%M:%S")
    file.write("\n\ndate and time = " + dt_string)
    #Writing the statistics to a file
    file.write("\nThe total number of wins is " + str(wins))
    file.write("\nThe total number of losses is " + str(losses))
    file.write("\nThe average number of rolls per win is " +
               str(winRolls / wins))
    file.write("\nThe average number of rolls per loss is " +
               str(lossRolls / losses))
    file.write("\nThe winning percentage is " + str(wins / number))
```

Lastly, we'll need to define the `main()` function as the point of entry. We'll have it play one game. Then the program will prompt the user to enter in the number of times to play the game and play it:

```
#The main function as the point of entry
def main():
    #Play one game
    print("Running one sample game in full:")
    playOneGame()

    #Play multiple games based on the entry by the user
    number = int(input("How many games would you want to have tested: "))
    playMultipleGames(number)
```

```
if __name__ == "__main__":
    main()
```

This last piece of code is slightly unique and new. Every Python module has its `__name__` automatically defined. If its name is set to `__name__`, it implies that the module is being run standalone by the user and we can perform a call to the `main()` function. If this module was imported by another module or .py file, the `__name__` is set to the name of the module so the file itself would not run.

KEY CONCEPT

__name__ represents the module name if it's not the entry point to the program, or it has a value of `__main__` if it is the entry point of the program.

There we are! The program as it is now is a functional one. With any program, we can always extend it further, but now it meets the program requirements. In the next lesson, we will be testing use cases on this program.

TRY IT

Directions: Finally add all new functions to the main.py file. The complete code for this "main" program is shown below.

```
#This game plays the game of dice called craps where
#Players would bet on the outcomes of a pair of dice rolls.
#If the sum of the dice is 2, 3 or 12, the player loses immediately.
#If the sum of the dice is 7 or 11, they win immediately.
#The purpose of the program is to simulate the results between two players

from player import Player

#Importing the datetime to get the current date and time
from datetime import datetime

#This function is created to play a single game and print out the results after each roll.
def playOneGame():

    player = Player()
    while not player.isWinner() and not player.isLoser():
        player.rollDice()
        print(player)
    if player.isWinner():
        print("You win!")
    else:
        print("You lose!")

#This function is created to play multiple games and outputs the results based
#on the number of games selected.
def playMultipleGames(number):
    #Initializing the variables
    wins = 0
    losses = 0
    winRolls = 0
    lossRolls = 0

    #Looping through the number of executions
    for count in range(number):
        player = Player()
        hasWon = player.play()
        rolls = player.getNumberOfRolls()
        if hasWon:
            wins += 1
            winRolls += rolls
        else:
            losses += 1
            lossRolls += rolls

    #Calculating the statistics
    print("The total number of wins is", wins)
    print("The total number of losses is", losses)
    print("The average number of rolls per win is %0.2f" % \
          (winRolls / wins))
    print("The average number of rolls per loss is %0.2f" % \
          (lossRolls / losses))
    print("The winning percentage is %0.3f" % (wins / number))
    print("The multi-game has been saved into the log.")

    logStats(wins, losses, winRolls, lossRolls, number)

#This function will log each multi game run to include the date/time and the details of the run
def logStats(wins, losses, winRolls, lossRolls, number):
    file = open("log.txt", "a")

    #Create a date/time object to get the current date and time
    now = datetime.now()
    #Formatting the output of the date and time to dd/mm/YY H:M:S
    dt_string = now.strftime("%d/%m/%Y %H:%M:%S")
    file.write("\n\ndate and time = " + dt_string)
    #Writing the statistics to a file
    file.write("\nThe total number of wins is " + str(wins))
    file.write("\nThe total number of losses is " + str(losses))
    file.write("\nThe average number of rolls per win is " +
               str(winRolls / wins))
    file.write("\nThe average number of rolls per loss is " +
               str(lossRolls / losses))
    file.write("\nThe winning percentage is " + str(wins / number))

#The main function as the point of entry
```

```
def main():
    #Play one game
    print("Running one sample game in full:")
    playOneGame()

    #Play multiple games based on the entry by the user
    number = int(input("How many games would you want to have tested: "))
    playMultipleGames(number)

if __name__ == "__main__":
    main()
```

TERM TO KNOW

`__name__`
`__name__` represents the module name if it's not the entry point to the program, or it has a value of `__main__` if it is the entry point of the program.

5. Guided Brainstorming

THINK ABOUT IT

Our final version of the demonstration program was likely more complex than our Sophia graders would expect to see. For that program, we wanted to use many of the syntax and concepts that we learned along the way. Remember your program can be as simple or complex as you feel it should be. The demonstration program had the added elements of splitting the program into modules for reuse and writing to files. As you develop your own program, try to keep it simple at the beginning. Once you're comfortable making the program more complex, you can add features for enhancement and reuse if desired.

Now back to our Drink Order program that we referenced from Unit 1:

EXAMPLE

```
#If drink is equal to water
#Ask user to enter in hot or cold
#Store input in heat
#If heat equal to hot
        #Add hot to outputString
        #Else If heat equal to cold
                #Add cold to outputString
                #Ask user to enter in ice or not
                #If ice is yes
                        #Add ice to outputString
                #Else
                        #Add no ice to output String
#Else If drink is equal to coffee
        #Ask user to enter in decaf or not
#Store input in decaf
#If decaf equal to Yes
        #Add decaf to outputString
        #Ask user to enter in Milk, cream or nonet
#Store input in milkCream
#If milkCream equal to milk
        #Add milk to outputString
        #Else If milkCream equal to cream
                #Add milk to outputString
        #Ask user to enter in sugar or not
#Store input in sugar
#If sugar equal to Yes
        #Add sugar to outputString
#Else If drink is equal to tea
        #Ask user to enter in teaType
#Store input in teaType
#If teaType equal to green
```

Our resulting program after the conversions without any other changes results in the following:

```
drinkDetails=""
drink = input('What type of drink would you like to order?\nWater\nCoffee\nTea\nEnter your choice: ')
if drink == "Water":
    drinkDetails=drink
    temperature = input("Would you like your water? Hot or Cold: ")
    if temperature == "Hot":
        drinkDetails += ", " + temperature
    elif temperature == "Cold":
        drinkDetails += ", " + temperature
        ice = input("Would you like ice? Yes or No: ")
        if ice == "Yes":
            drinkDetails += ", Ice"
    else:
        drinkDetails += ", unknown temperature entered."
elif drink == "Coffee":
    drinkDetails=drink
    decaf = input("Would you like decaf? Yes or No: ")
    if decaf == "Yes":
        drinkDetails += ", Decaf"
    milkCream = input("Would you like Milk, Cream or None: ")
    if milkCream == "Milk":
        drinkDetails += ", Milk"
    elif milkCream == "Cream":
        drinkDetails += ", Cream"
    sugar = input("Would you like sugar? Yes or No: ")
    if sugar == "Yes":
        drinkDetails += ", Sugar"
elif drink == "Tea":
    drinkDetails=drink
    teaType = input("What type of tea would you like? Black or Green: ")
    if teaType == "Black":
        drinkDetails += ", " + teaType
    elif teaType == "Green":
        drinkDetails += ", " + teaType
else:
    print("Sorry, we did not have that drink available for you.")
print("Your drink selection: ",drinkDetails)
```

Be sure as you start your development that you're including all of the core functionality that you need from the beginning. Without doing so, you're going to get into some issues along the way that can be harder for you to decipher as the program gets more complex. As you code your program, think of the following questions:

- Does the program work as intended? Are you seeing the expected output?
- Does the program implement all of the requirements from the client?
- Can the program be optimized using conditional statements, loops, functions, and classes?

This will ensure that you're meeting all of the requirements as you progress in programming your solution.

TRY IT

Directions: At this point, you should be building out your program to put it into a state that is testable to ensure that everything is working as expected. For any areas that you have not implemented yet, it would be a good idea to comment those sections out as needed.

SUMMARY

In this lesson, we broke down the demonstration program based on **core functionalities**. We wanted the ability to be able to reuse some of the functionality for other cases. To do that, we created the **die** and **player** modules that allow us to import features from them into the main program. We returned to the main program and added two play functions that allow for a single and multiple games. To **finish the main program**, we added a function to output the results to a file. In the **Guided Brainstorming** section, we took the pseudocode of the Drink Order program and converted that to code. In the next lesson, we will try testing these programs.

Best of luck in your learning!

TERMS TO KNOW

__name__

`__name__` represents the module name if it's not the entry point to the program, or it has a value of `__main__` if it is the entry point of the program.

__str__

The `__str__` method is used for classes and returns back a string value that the developer defines.