Testing As You Go

by Sophia

WHAT'S COVERED

In this lesson, we will learn about testing as we progress. Specifically, this lesson covers:

1. Checking for Errors

Errors, bugs, and exceptions are always there in any program that we develop. These errors and issues can cause all types of problems with the program. As we work through the code, it's a good idea to test the program as much as we can. We want to try to do everything that we can to break it and then find ways to help protect it from breaking in the future. Some examples of things we should test and debug are:

1. Invalid inputs – if the program asks the user to enter in an integer, try to enter something else, like a character.
2. Edge and corner cases for conditional statements.
3. Uppercase and lowercase inputs for conditional statements compared to strings.
4. Complete testing of end-user use or the program from start to finish.

There may be times while debugging code when we cannot find a solution for an error. If so, we can move on temporarily, and we may simply add the code into a comment area and test it again later on.
In our program, there is only a single required input, which simplifies things a bit. We are prompted to enter a number. What happens if we enter a character?

Oops, we encounter an error:

```
Running one sample game in full:
(3, 4) total = 7
You win!
How many games would you want to have tested: a
Traceback (most recent call last):
  File "/home/main.py", line 86, in <module>
    main()
  File "/home/main.py", line 82, in main
```

```
    number = int(input("How many games would you want to have tested: "))
ValueError: invalid literal for int() with base 10: 'a'
```

TRY IT

Directions: Try running the demonstration program and entering a letter instead of a number.

This is because we didn't implement exception handling yet. We originally had:

EXAMPLE

```
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)
```

We'll make some changes to handle the exception. First, we will set `number` to 0 and create a new variable to accept the user's input. Then, we will add a `try and except` statement which will ensure the input is an integer and if not, will print a statement to notify the user.

```
def main():
    #Play one game
    print("Running one sample game in full:")
    playOneGame()
    number = 0
    #Play multiple games based on the entry by the user
    user_input = input("How many games would you want to have tested: ")

    try:
      number = int(user_input)
      playMultipleGames(number)
    except ValueError:
      print("Please enter in a number for the number of games.")
```

When this is executed, here is the updated output screen.

```
Running one sample  game in full:
(5, 6) total = 11
You win!
How many games would you want to have tested: a
Please enter in a number for the number of games.
```

TRY IT

Directions: Add the `try and except` statement and try this again using an invalid entry.

It is good that there was no error but we haven't solved the issue. The program just ended. Ideally, we want to have the user prompted again to give an input until a valid number has been entered. By adding in a `while` loop while `number` is equal to

zero, we can continue asking the user until a number is set.

EXAMPLE

```
def main():
    #Play one game
    print("Running one sample game in full:")
    playOneGame()
    number = 0
    #Play multiple games based on the entry by the user
    while number == 0:
      user_input = input("How many games would you want to have tested: ")
      try:
        number = int(user_input)
        playMultipleGames(number)
      except ValueError:
        print("Please enter in a number for the number of games.")
```

Now our output screen can continue to ask for input.

```
Running one sample  game in full:
(4, 5) total = 9
(6, 3) total = 9
You win!
How many games would you want to have tested: 5
The total number of wins is 2
The total number of losses is 3
The average number of rolls per win is 1.50
The average number of rolls per loss is 4.33
The winning percentage is 0.400
The multi-game has been saved into the log.
```

TRY IT

Directions: Now add the `while` loop and try again.

This is great for a larger number where the total wins and total losses will probably be greater than zero. What happens if the number of games doesn't satisfy this, and one of them is zero?

```
Running one sample  game in full:
(4, 2) total = 6
(5, 3) total = 8
(5, 5) total = 10
(3, 2) total = 5
(5, 3) total = 8
(5, 5) total = 10
(5, 6) total = 11
(4, 3) total = 7
You lose!
```

```
How many games would you want to have tested: 1
The total number of wins is 1
The total number of losses is 0
The average number of rolls per win is 2.00
Trackback (most recent call last):
  File "main.py", line 95, in <module>
   main()
  File "main.py", line 89, in main
   playMultipleGames(number)
  File "main.py", line 52, in playMultipleGames
   (lossRolls / losses))
ZeroDivisionEror: division by zero
```

We ran the game once—what happened here? We won once and then ended the program. However, we're getting an error due to a division by zero. It's taking the number of rolls and dividing it by zero losses. This causes an error. We need to change the code to handle those situations where either the losses or the wins are equal to zero. We'll need to do that in both the `playMultipleGames` and the `logStats` functions:

EXAMPLE

```
#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)
    if wins > 0:
      print("The average number of rolls per win is %0.2f" % \
          (winRolls / wins))
    if losses > 0:
      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)


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))
    if wins > 0:
      file.write("\nThe average number of rolls per win is " +
                 str(winRolls / wins))
    if losses > 0:
      file.write("\nThe average number of rolls per loss is " +
                 str(lossRolls / losses))
    file.write("\nThe winning percentage is " + str(wins / number))
```

With those changes, let's try that again.

```
Running one sample  game in full:
(1, 5) total = 6
(5, 5) total = 10
(4, 3) total = 7
You lose!
How many games would you want to have tested: 1
The total number of wins is 1
The total number of losses is 0
The average number of rolls per win is 3.00
The winning percentage is 1.000
The multi-game has been saved into the log.
```

Now that makes more sense. If there are no losses, there should be no average number of rolls per loss output to the screen. Likewise for wins:

```
Running one sample  game in full:
(3, 1) total = 4
(3, 3) total = 6
(4, 3) total = 7
You lose!
```

```
How many games would you want to have tested: 1
The total number of wins is 0
The total number of losses is 1
The average number of rolls per win is 1.00
The winning percentage is 0.000
The multi-game has been saved into the log.
```

TRY IT

Directions: Add the changes made to the `playMultipleGames` and `logStats` functions and run the game again. See if you get the same output when inputting 1 game to play.

Now we have that tested. Let's test an edge case of entering a negative number or a zero. Neither one of those should be permitted.

```
Running one sample  game in full:
(3, 1) total = 4
(3, 3) total = 6
(5, 3) total = 8
(5, 2) total = 7
You lose!
How many games would you want to have tested: 0
The total number of wins is 0
The total number of losses is 0
Traceback (most recent call last):
  File "/home/main.py", line 99, in <module>
   main()
  File "/home/main.py", line 93, in main
   playMultipleGames(number)
  File "/home/main.py", line 55, in playMultipleGames
   print("The winning percentage is %0.3f" % (wins / number))
ZeroDivisionEror: division by zero
```

Let's handle that within the `try and except` statement after we've converted the `user_input` to an `int`. If it's equal to or less than zero, we'll let the user know to make that change:

EXAMPLE

```
#The main function as the point of entry
def main():
    #Play one game
    print("Running one sample game in full:")
    playOneGame()
    number = 0
    #Play multiple games based on the entry by the user
    while number <= 0:
      user_input = input("How many games would you want to have tested: ")
      try:
        number = int(user_input)
        if number <=0:
```

```
            print("Please enter in a positive number.")
            number = 0
          else:
            playMultipleGames(number)
        except ValueError:
          print("Please enter in a number for the number of games.")
```

Now let's try that again with an edge case.

```
Running one sample  game in full:
(4, 2) total = 6
(2, 5) total = 7
You lose!
How many games would you want to have tested: 0
Please enter in a positive number.
How many games would you want to have tested: -1
Please enter in a positive number.
How many games would you want to have tested:
```

That's much better. The program can correctly respond to the user input. We can now move on to any other testing that needs to be performed on our program.

TRY IT

Directions: Add the `try and except` statement and test the game again. This time any invalid output should be handled.

HINT

Debugging

If you have errors or issues that you can't figure out, remember you have access to the IDE debugger. Back in Unit 1's Debugging Conditional Statements and Unit 3's Finishing the Tic-Tac-Toe lessons, we set up breakpoints and tracked specific variables as they changed within the code. This will help you to troubleshoot those problems and ensure that the variables are being set correctly.

---

2. Adding Our Fourth Journal Entry

THINK ABOUT IT

At this point, we are ready to add the next journal entry (Part 4) for the Touchstone. The testing of your program is something you want to not only perform but also show what you did for it and show the fixes that you made to resolve the problems that occurred.

What we would not want to add to our journal entry are the details of what was wrong without explaining what the issue was and how we fixed the issues. For example, this would be a bad entry for Part 4.

Bad Example of Journal Entry for Part 4

Here is my test of the casino craps game.

```
Running one sample  game in full:
```

```
(5, 6) total = 11
You win!
How many games would you want to have tested: a
Please enter in a number for the number of games.
```

A better entry for Part 4 would explain the errors and issues that came up and what was done to fix them. Simply looking at this output statement, we can't identify what the issue was specifically, which in this case was that the user entered the letter "a" for the input.

Good Example of Journal Entry for Part 4

After changing my initial code in the main program to check for invalid user entries by adding the try and except statements, I noticed it handled the invalid input (no error) but would stop after that and not request another entry from the user. Here was my initial code:

```
def main():
    #Play one game
    print("Running one sample game in full:")
    playOneGame()
    number = 0
    #Play multiple games based on the entry by the user
    user_input = input("How many games would you want to have tested: ")

    try:
      number = int(user_input)
      playMultipleGames(number)
    except ValueError:
      print("Please enter in a number for the number of games.")
```

And the output screen did handle the exception of the letter "a" as input but the program stopped. The output screen looks like this.

```
Running one sample  game in full:
(5, 6) total = 11
You win!
How many games would you want to have tested: a
Please enter in a number for the number of games.
```

I want the program to prompt the user again to enter another input until a valid number has been entered. I then added a `while` loop. By using this loop, this will repeat while the variable `number` is equal to zero. The loop will continue to ask the user for a value until a valid number is set. I changed my code to look like this:

```
def main():
    #Play one game
    print("Running one sample game in full:")
    playOneGame()
    number = 0
    #Play multiple games based on the entry by the user
    while number == 0:
```

```
      user_input = input("How many games would you want to have tested: ")
      try:
        number = int(user_input)
        playMultipleGames(number)
      except ValueError:
        print("Please enter in a number for the number of games.")
```

Now after adding in the `while` loop, I ran the code again and entered in the letter “a” as input. When doing so, the program then prompted me again to enter in the number of games rather than ending the program. This solves the issue of the program ending now.

```
Running one sample  game in full:
(5, 4) total = 9
(2, 1) total = 3
(1, 1) total = 2
(4, 5) total = 9
You win!
How many games would you want to have tested: a
Please enter in a number for the number of games.
How many games would you want to have tested:
```

If we preview the Example Python Journal Submission document, we see this was added as the entry to Part 4.

With this example, we’re now being prompted every time an invalid number is entered in. This would be documentation for the testing of one issue.

---

3. Guided Brainstorming

You’ll notice that as you program and test, you will likely change the code as you go. Doing so will ensure that the program works correctly every step of the way and simplifies debugging. Every program will be different so it is important that you’re thinking about the problem. Let’s look at our Drink Order program again:

```
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)
```

We should test each of the branches and scenarios of the valid values between the water, coffee, and tea to ensure it works as expected:

```
What type of drink would you like to order?
Water
Coffee
Tea
Enter your choice: Water
Would you like your water? Hot or Cold: Hot
Your drink selection: Water, Hot

What type of drink would you like to order?
Water
Coffee
Tea
Enter your choice: Water
Would you like your water? Hot or Cold: Cold
Would you like ice? Yes or No: Yes
Your drink selection: Water, Cold, Yes


What type of drink would you like to order?
Water
Coffee
Tea
```

```
Enter your choice: Coffee
Would you like decaf? Yes or No: Yes
Would you like Milk, Cream or None: Cream
Would you like sugar? Yes or No: Yes
Your drink selection: Coffee, Decaf, Cream, Sugar

What type of drink would you like to order?
Water
Coffee
Tea
Enter your choice: Tea
What type of tea would you like? Black or Green: Black
Your drink selection: Tea, Black
```

This is great and all but what happens in the case where you enter input that doesn't work? For example, let's try entering water in lowercase instead of the capital "W" that the input was expecting.

```
What type of drink would you like to order?
Water
Coffee
Tea
Enter your choice: water
Sorry, we did not have that drink available for you.
Your drink selection:
```

Well, that didn't quite work out as the check was case-sensitive. If we would want to do the check without the case sensitivity, there are different approaches we could do, including converting the input and result to lowercase or uppercase to make the comparison.

What happens when we enter an input that wasn't expected:

```
What type of drink would you like to order?
Water
Coffee
Tea
Enter your choice: soda
Sorry, we did not have that drink available for you.
Your drink selection:
```

The choice of soda was not an option that we had defined but how the program should react to that input may be different, so you need to define how the program should handle those issues. Perhaps it should do nothing or perhaps it needs to continuously ask the user for valid input.

Directions: Needless to say, you should be testing your code as you are writing, but there are a lot of logical issues that can occur, especially with mistakes that may not be caught right away. It's easier to test cases that are supposed to work as intended versus ones that may break the code. For example, if the program is asking for a number, we could test with a letter or if a program is asking for numbers 1 through 10, we enter 0, 11, or -5. Take the time to ensure that you have tested those

edge cases as well as the completely incorrect cases. Review the example of a good entry for Part 4 in the Example Python Journal Submission document and add your entry for Part 4 to your Python Journal.

SUMMARY

In this lesson, we tested our demonstration program by **checking for errors**. We first checked for invalid user inputs and needed to add some exception handling. Then we noticed that the program did not repeat the input request after an invalid value. We needed to add a loop to continue the input request until a valid value was added. We then had an opportunity to see a good example for **our fourth journal entry**. This good example had both what we were testing for and how we solved the need. In the **Guided Brainstorming** section we identified another example of good test cases to ensure the program is working the way it is intended to.

Best of luck in your learning!