# Section 12

Debugging

Mohammed Asir Shahid

2021-08-04

## Contents



## 1 The raise and assert Statements

Now we might start finding some more complicated bugs. Debugging is a tool that we need to learn in order to find what we did wrong.

For example, a zero divide error occurs when we try to divide a number by 0. We learned how to handle exceptions with try and except statements to deal with expected errors.

## 1.1 Raising Your Own Exceptions

We can also raise our own exceptions in our code. This is a way of saying that Python should stop running the code in this function and move to the except statement.

We can raise exceptions using the raise statement.

```
raise Exception("This is the error message.")
```

## 1.2 Box Example

We want to create a function that creates a box using some supplied characters.

```
def boxPrint(symbol, width, height):

    if len(symbol)!=1:
        raise Exception("symbol needs to be of length 1")

    if width < 2 or height < 2:
        raise Exception("width and height must be greater than or equal to 2")

    print(symbol*width)

    for i in range(height-2):
        print(symbol + (" "*(width-2)) + symbol)

    print(symbol*width)

boxPrint("*", 15, 5)
boxPrint("O", 5, 20)
```

```
***************
*             *
*             *
*             *
***************
```

```
OOOOO
O   O
O   O
O   O
O   O
O   O
O   O
O   O
O   O
O   O
O   O
O   O
O   O
O   O
O   O
O   O
O   O
O   O
O   O
OOOOO
```

Our error message is called a traceback. This is because it shows information showing where the error occurred.

## 2 The traceback.format$_{\mathbf{exc}}$() Function

We can get the traceback error text as a string vale using this function.

```
import traceback

try:
    raise Exception("This is the error message.")
except:
    errorFile=open("error_log.txt","a")
    errorFile.write(traceback.format_exc())
    errorFile.close()
    print("The traceback info was written error_log.txt")

error=open("error_log.txt")
```

```
print(error.read())
```

```
The traceback info was written error_log.txt
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
Exception: This is the error message.
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
Exception: This is the error message.
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
Exception: This is the error message.
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
Exception: This is the error message.
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
Exception: This is the error message.
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
Exception: This is the error message.
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
Exception: This is the error message.
```

## 2.1 Assertions and the assert Statement

An assertion is a sanity check that makes sure the code isn't doing something really wrong. These are performed by assert statements. If the sanity check fails, then an assertion error exception is raised. These assertions are for programmer errors and the program should not run after an assertion is raised.

```
assert False, "This is the error message"
```

Let's try to create a simple traffic simulator program with intersections with stop lights.

```
market_2nd={"ns": "green", "ew": "red"}

def switchLights(intersection):
    for key in intersection.keys():
        print(intersection[key])
        if intersection[key]=="green":
            intersection[key]="yellow"
        elif intersection[key]=="yellow":
            intersection[key]="red"
        elif intersection[key]=="red":
            intersection[key]="green"
    assert "red" in intersection.values(), "Neither light is red!"

print(market_2nd)
switchLights(market_2nd)
print(market_2nd)
```

# 3 Logging

Logging is similar to putting a print function in your code to output variables values while the program is running to debug.

Python has a logging module to make debugging like this and creating a record of custom messages that you write easier.

## 3.1 The logging.basicConfig() Function

```
import logging

logging.basicConfig(level=logging.DEBUG, format="%(asctime)s - %(levelname)s - %(messag
```

## 3.2 logging.debug() Function

```
import logging

logging.basicConfig(level=logging.DEBUG, format="%(asctime)s - %(levelname)s - %(messag
```

```
logging.disable(logging.CRITICAL)

logging.debug("Start of program")

def factorial(n):
    logging.debug("Start of factorial (%s)" % (n))
    total=1
#    for i in range(0,n+1):
    for i in range(1,n+1):
        total*=i
        logging.debug("i is %s, total is %s" % (i,total))
    logging.debug("Return value is %s" % (total))
    return total

print(factorial(5))

logging.debug("End of program")
```

```
120
```

Using debugging, we can see that since range begins at 0, we do 0*1 and make the total 0. Then anything we multiply by 0 and get 0 for the rest.

Why should we use this instead of print? If we were doing this with the print function, we'd have to find all the print statements and then delete them manually. That can be time consuming and we might accidentally delete a print call that we want to keep. Instead if we use the debug function in the logging module, we can simply turn off the logging message by calling logging.disable.

### 3.3 Log Levels

We have five different log levels. In order of ascension, they are:

1. debug
2. info
3. warning
4. error

5. critical

When we call logging.debug(), we are creating a logging message at the debug level. There is also logging.info, warning, error, and critical. Since we passed in logging.CRITICAL into the above logging.disable() function, it disabled all logging messages at the critical level or lower.

When debugging our program, we can call different logging functions based on their priority. If something is not that important, then we can keep it at debug, but if there is something more important then we can do warning, error, or critical.

### 3.4 Logging to a Text File

If we want to write the logging messages to a file, then we can change the basicConfig that we used in the beginning. We can add a filename argument and set it equal to the name of the file we want to write to. Then there will be no logging messages on the screen, just in the file.

```
import logging

logging.basicConfig(filename="myProgrammingLog.txt",level=logging.DEBUG, format="%(asct
```

## 4 Using the Debugger

Debuggers are features that let you examine your code one line at a time. The debugger will run a single line of code and then wait for you to tell it to continue. The debugger works differently on all text editors/IDEs.