Chapter 4 Debugging

3.4.1 Raising Exceptions

Python raises an exception whenever it tries to execute invalid code.Raising an exception is a way of saying, “Stop running the code in this function and move the program execution to the except statement.” Exceptions are raised with a raise statement. In code, a raise statement consists of the following:\

- The raise keyword
- A call to the Exception() function
- A string with a helpful error message passed to the Exception() function
  For example:

In [ ]:

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

If there are no try and except statements covering the raise statement that raised the exception, the program simply crashes and displays the exception’s error message.

In [ ]:

```
def boxPrint(symbol, width, height):
    if len(symbol) != 1:
      raise Exception('Symbol must be a single character string.')
    if width <= 2:
       raise Exception('Width must be greater than 2.')
    if height <= 2:
       raise Exception('Height must be greater than 2.')

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

for sym, w, h in (('*', 4, 4), ('O', 20, 5), ('x', 1, 3), ('ZZ', 3, 3)):
    try:
        boxPrint(sym, w, h)
    except Exception as err:
        print('An exception happened: ' + str(err))
```

3.4.2 Getting the Traceback as a String

When Python encounters an error, it produces a treasure trove of error information called the traceback. The traceback includes the error message, the line number of the line that caused the error, and the sequence of the function calls that led to the error. This sequence of calls is called the call stack.

In [ ]:

```
def spam():
    bacon()

def bacon():
    raise Exception('This is the error message.')

spam()
```

Python displays the traceback whenever a raised exception goes unhandled. But you can also obtain it as a string by calling traceback.format_exc(). This function is useful if you want the information from an exception's traceback but also want an except statement to gracefully handle the exception.

In [ ]:

```
import traceback
try:
        raise Exception('This is the error message.')
except:
        errorFile = open('errorInfo.txt', 'w')
        errorFile.write(traceback.format_exc())
        errorFile.close()
        print('The traceback info was written to errorInfo.txt.')
```

3.4.3 Assertions

An assertion is a sanity check to make sure your code isn't doing something obviously wrong. These sanity checks are performed by assert statements. If the sanity check fails, then an AssertionError exception is raised. An assert statement consists of the following:

- The assert keyword
- A condition (that is, an expression that evaluates to True or False)
- A comma
- A string to display when the condition is False

In [ ]:

```
ages = [26, 57, 92, 54, 22, 15, 17, 80, 47, 73]
ages.sort()
ages
```

In [ ]:

```
assert ages[0] <= ages[-1]
ages.reverse()
ages
```

In [ ]:

```
assert ages[0] <= ages[-1]
```

3.4.3.1 Using an Assertion in a Traffic Light Simulation

The data structure representing the stoplights at an intersection is a dictionary with keys 'ns' and 'ew', for the stoplights facing north-south and east-west, respectively. The values at these keys will be one of the strings 'green', 'yellow', or 'red'.

In [ ]:

```
market_2nd = {'ns': 'green', 'ew': 'red'}
mission_16th = {'ns': 'red', 'ew': 'green'}
```

In [ ]:

```
def switchLights(stoplight):
    for key in stoplight.keys():
        if stoplight[key] == 'green':
            stoplight[key] = 'yellow'
        elif stoplight[key] == 'yellow':
            stoplight[key] = 'red'
        elif stoplight[key] == 'red':
            stoplight[key] = 'green'
    assert 'red' in market_2nd.values(), 'Neither light is red! ' + str(market_2nd)

switchLights(market_2nd)
```

3.4.4 Logging

If you've ever put a print() statement in your code to output some variable's value while your program is running, you've used a form of logging to debug your code.Logging is a great way to understand what's happening in your program and in what order it's happening. Python's logging module makes it easy to create a record of custom messages that you write. These log messages will describe when the program execution has reached the logging function call and list any variables you have specified at that point in time.

3.4.4.1 Using the logging Module

To enable the logging module to display log messages on your screen as your program runs, copy the following to the top of your program.

In [2]:

```
import logging
logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s -  %(levelname)s -  %(message
logging.debug('Start of program')

def factorial(n):
    logging.debug('Start of factorial(%s%%)'  % (n))
    total = 1
    for i in range(1,n + 1):
        total *= i
        logging.debug('i is ' + str(i) + ', total is ' + str(total))
    logging.debug('End of factorial(%s%%)'  % (n))
    return total

print(factorial(5))
logging.debug('End of program')
```

```
 2020-08-29 12:11:29,061 -  DEBUG -  Start of program
 2020-08-29 12:11:29,063 -  DEBUG -  Start of factorial(5%)
 2020-08-29 12:11:29,064 -  DEBUG -  i is 1, total is 1
 2020-08-29 12:11:29,064 -  DEBUG -  i is 2, total is 2
 2020-08-29 12:11:29,065 -  DEBUG -  i is 3, total is 6
 2020-08-29 12:11:29,066 -  DEBUG -  i is 4, total is 24
 2020-08-29 12:11:29,067 -  DEBUG -  i is 5, total is 120
 2020-08-29 12:11:29,068 -  DEBUG -  End of factorial(5%)
 2020-08-29 12:11:29,069 -  DEBUG -  End of program

120
```

3.4.4.2 Don’t Debug with the print() Function

Log messages are intended for the programmer, not the user. The user won’t care about the contents of some dictionary value you need to see to help with debugging; use a log message for something like that. For messages that the user will want to see, like File not found or Invalid input, please enter a number, you should use a print() call.

3.4.4.3 Logging Levels

In [3]:

```
import logging
logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s -%(levelname)s -  %(message)s
logging.debug('Some debugging details.')
logging.info('The logging module is working.')
logging.warning('An error message is about to be logged.')
logging.error('An error has occurred.')
logging.critical('The program is unable to recover!')
```

```
 2020-08-29 12:24:18,551 -  DEBUG -  Some debugging details.
 2020-08-29 12:24:18,552 -  INFO -  The logging module is working.
 2020-08-29 12:24:18,553 -  WARNING -  An error message is about to be logge
d.
 2020-08-29 12:24:18,554 -  ERROR -  An error has occurred.
 2020-08-29 12:24:18,555 -  CRITICAL -  The program is unable to recover!
```

3.4.4.4 Disabling Logging

After you've debugged your program, you probably don't want all these log messages cluttering the screen. The logging.disable() function disables these so that you don't have to go into your program and remove all the logging calls by hand. Pass logging.disable() a logging level, and it will suppress all log messages at that level or lower.to disable logging entirely, just add logging.disable(logging.CRITICAL) to your program.

In [4]:

```
 import logging
logging.basicConfig(level=logging.INFO, format=' %(asctime)s -%(levelname)s -  %(message)s'
logging.critical('Critical error! Critical error!')
logging.disable(logging.CRITICAL)
logging.critical('Critical error! Critical error!')
logging.error('Error! Error!')
```

```
 2020-08-29 12:29:03,056 -  CRITICAL -  Critical error! Critical error!
```

3.4.4.5 Logging to a File

Instead of displaying the log messages to the screen, you can write them to a text file. The logging.basicConfig() function takes a filename keyword argument

In [6]:

```
import logging
logging.basicConfig(filename='myProgramLog.txt', level=logging.DEBUG, format='%(asctime)s -
#The log messages will be saved to myProgramLog.txt.
```

3.4.5 Mu's Debugger

The debugger is a feature of the Mu editor, IDLE, and other editor software that allows you to execute your program one line at a time. The debugger will run a single line of code and then wait for you to tell it to continue. By running your program "under the debugger" like this, you can take as much time as you want to examine the values in the variables at any given point during the program's lifetime. This is a valuable tool for tracking down bugs.