Chapter 6 :
Testing & Debugging

Aviral Janveja

1 Testing

We have written many programs in this course already, mostly to demonstrate a particular concept or for personal learning. However, when developing software in a professional setting for some client, it is imperative that we provide them with a well functioning, error free and robust software solution.

This is where the concept of testing and debugging comes in. It helps us make sure, that our code satisfies all the requirements, runs error free and is able to handle exceptional user inputs and edge-cases as well. The first step in this process, is to set yourself up for easy testing and debugging by writing clean, modular and well documented code.

1.1 Testing Approaches

Software testing can be majorly classified into two high level approaches :

1.1.1 Black Box Testing

Black box testing is a testing technique in which the internal workings of the code are not known to the tester. The tester only focuses on the input and output of the program.

1.1.2 White Box Testing

White box testing is a testing technique in which the tester has knowledge of the internal workings of the code and can test individual code snippets, algorithms and functions.

1.2 Testing Classes

There are three general classes of testing that you can perform :

1.2.1 Unit Testing

Unit tests focus on validating small testable parts of an application. In context of Python, unit tests make sure that individual functions run according to their specifications.

1.2.2 Regression Testing

Regression testing aims to ensure that recent changes or enhancements to an application do not introduce new bugs or break existing functionality.

1.2.3 Integration Testing

Integration testing examines the interaction between different components, modules or systems to ensure that they work together as intended.

1.3 Common Error Messages

Before moving on to debugging, let us first examine some common error messages and their meanings.

1.3.1 SyntaxError

This error occurs when the Python interpreter encounters code that doesn't follow the correct syntax rules. It could be a missing parenthesis, a misplaced colon, or some other syntax-related issue.

1.3.2 NameError

These types of errors can result from attempting to use a variable or function that hasn't been defined. Make sure to check for typos in variable or function names, and make sure they are defined before use.

1.3.3 IndexError

This type of error happens when you're trying to access an index in a sequence, like a list or a string that doesn't exist. To fix it, make sure that the index being used is within the valid range of the sequence.

1.3.4 TypeError

This is a common exception in Python that occurs when an operation or function is applied to an object of an inappropriate type. For example, attempting to add a string and an integer or multiply a list by a string or calling a function with an incorrect number of arguments.

1.3.5 ValueError

It is a type of error that occurs when a function receives an argument of the correct type but an inappropriate value. For example, when trying to convert a string into an integer. It works perfectly fine when we pass '123' (a string that can be converted to an integer). However, when we pass 'abc' (a string that cannot be converted into an integer), it throws a ValueError.

2 Debugging

Now that we understand some common error types, let's explore various techniques and tools that can help you debug your Python code efficiently.

2.1 Print Statements

It is essential to understand how your code is executing and the values of variables at different points in the program. By strategically placing print statements at different parts of your code, you can create a log of sorts that shows the order in which different sections of your code are being executed. This can help you understand the control flow and pinpoint where the program might be deviating from your expectations. Checkout the following example that showcases the use of print statement for debugging : debug_print.py.

While print statements are often the quickest and most straightforward way to get a glimpse into a program's execution flow, especially during initial development, they can be cumbersome to manage and may not be appropriate for production code and therefore comes Logging, which provides a structured way to record information.

2.2 Logging

Logging is like writing notes while your program runs. Instead of just printing things to the screen, you write them to a log. It helps you keep track of what your program is doing, especially when things go wrong.

Logging can be configured to write messages to both the console and a file named example.log. The "format" parameter can be used to customize the appearance of log messages, including the timestamp, log level, and the actual log message. The following program showcases how logging could be utilized for a scalable debugging approach : debug_logging.py.

Further, Python development environments many a times provide inbuilt logging and debugging tools that can help make the debugging process easier. It is possible to set the log level based on severity and customize the log messages as well.

2.3 Exception Handling

An error is also called an exception, as it is an exception to what the programmer expected.

2.3.1 try-except

The **try-except** statement is a way to handle exceptions in Python. It prevents your program from crashing abruptly, allowing you to gracefully handle errors and log relevant information. For example :

```
try:
    a = int(input("Enter a number: "))
    b = int(input("Enter another number : "))
    print("Sum = ", a + b)
```

```
        print("Division = ", a / b)

    except:
        print("Invalid Input!")
```

The **try** block contains the code that might raise and exception. The **except** block contains the code that is executed if an exception occurs in the try block.

The above is a general except statement that can be used to catch any exception. You can also catch specific exceptions by specifying separate except clauses to deal with different types of exceptions. The following program demonstrates the same : specific_exception.py

2.3.2 finally

We can also have a **finally** block that is always executed, regardless of whether an exception occurred.

```
try:
    x > 3
except:
    print("Something went wrong")
finally:
    print("The try-except block is finished")
```

It is used for executing clean-up code that should be run no matter what else happened, for example, closing a file.

2.3.3 raise

It is also possible to raise your own exception using the **raise** keyword. For example :

```
x = "hello"

if type(x) is not int:
raise TypeError("Only integers are allowed")
```

2.4 Assertions

The assert keyword lets you test if a condition in your code returns True. If not, the program will raise an AssertionError. You can add a message to be displayed, if the code returns False:

```
x = "welcome"

#if condition returns False, AssertionError is raised:
assert x == "hello", "x should be 'hello'"
```

Assert is useful in checking if the pre-conditions and post-conditions on your functions are exactly as you want them to be. So, as soon as an assert becomes false, the function is going to terminate immediately. This is useful as it prevents the program from propagating bad values forward. Thus, helping in spotting bugs as soon as they are introduced and where they are introduced.