Unit VI: Efficient Testing and Debugging

Now that you know how to write a code, you may have noticed that programming is not only about writing a perfect code for a particular problem, but it is also about finding not-so-simple bugs/errors in them. Syntax means the arrangement of letters and symbols in code. So if you get a syntax error, it usually means you've misplaced a symbol or letter somewhere.

Eg:
```
while True print('Hello')
SyntaxError: invalid syntax
```

In the example, the error is detected at the function print(), since a colon ':' is missing before it.

Debugging

It is the process using which we can fix any bugs/errors that are present in the program. The process involves, firstly identifying the error, then analyzing it and lastly fixing that bug. This process is considered to be extremely tedious and complex because we need to identify and resolve errors present in every stage of debugging. However, exception handling plays a pivotal role in debugging.

Exceptions and Exception handling, try-catch-throw, multiple catch, catch all, rethrowing exception, implementing user defined exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions.

Eg:
```
print(1/0)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    print(1/0)
ZeroDivisionError: division by zero
```

In the above example, the string printed as the exception type called 'ZeroDivisionError' is the name of the built-in exception that occurred. The preceding part of the error message shows the context where the exception occurred.

Some of the common built-in exceptions in python are:

1. SyntaxError: This exception is raised when the interpreter encounters a syntax error in the code, such as a misspelled keyword, a missing colon, or an unbalanced parenthesis.
2. TypeError: This exception is raised when an operation or function is applied to an object of the wrong type, such as adding a string to an integer.

3. NameError: This exception is raised when a variable or function name is not found in the current scope.
4. IndexError: This exception is raised when an index is out of range for a list, tuple, or other sequence types.
5. KeyError: This exception is raised when a key is not found in a dictionary.
6. ValueError: This exception is raised when a function or method is called with an invalid argument or input, such as trying to convert a string to an integer when the string does not represent a valid integer.
7. AttributeError: This exception is raised when an attribute or method is not found on an object, such as trying to access a non-existent attribute of a class instance.
8. IOError: This exception is raised when an I/O operation, such as reading or writing a file, fails due to an input/output error.
9. ZeroDivisionError: This exception is raised when an attempt is made to divide a number by zero.
10. ImportError: This exception is raised when an import statement fails to find or load a module.

You can write a program to handle selected exceptions which is known as user-defined exceptions.

1. Try and Except Statements:
   This is used to catch and handle exceptions in Python. ***Try block:*** Statements that can raise exceptions are kept inside the try clause. If no exception occurs, the except clause is skipped and execution of the try statement is finished.
   Except block: Statements that handle the exception occurred in the try clause are written inside the except clause.

```
try:
    # write your code that may raise an exception in here
    pass
except:
    pass
    # write code that will run if an exception is raised in the try block
```

Eg:

```
x, y = 10, 0
try:
    print(x/y)
except:
    print('Error: division by zero')
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
Error: division by zero
```

A try statement can have more than one except clause, to specify handlers for different exceptions. Please note that at most one handler will be executed catching specific exceptions in the Python.

Eg:

```
1   x= 10
2   y = "Hello"
3   try:
4       print(x/y)
5   except TypeError:
6       print(' Type Error: Cannot divide number by string')
7   except ZeroDivisionError:
8       print('Zero Division Error: Cannot divide by zero')
9   except:
10      print('Unknown Error')
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

Type Error: Cannot divide number by string
```

2. Try with Else Clause:
   The code enters the else block only if the try clause does not raise an exception.

```
try:
    # write your code that may raise an exception in here
    pass
except:
    pass
    # write code that will run if an exception is raised in the try block
else:
    pass
    # write code that will run if no exception is raised in the try block
```

Eg:

```
x = 10
y = 2
try:
    z = x/y
except TypeError:
    print(' Type Error: Cannot divide number by string')
except ZeroDivisionError:
    print('Zero Division Error: Cannot divide by zero')
except:
    print('Unknown Error')
else:
    print(z)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
5.0
```

3. Try with Finally Clause:
   A block with Finally is always executed after the try and except blocks i.e. block always executes after the normal termination of the try block or after the try block terminates due to some exception.

```
try:
    # write your code that may raise an exception in here
    pass
except:
    pass
    # write code that will run if an exception is raised in the try block
else:
    pass
    # write code that will run if no exception is raised in the try block
finally:
    # write code that will run regardless of whether an exception is raised or not
```

Eg:

```
1   x = 10
2   y = 2
3   try:
4       z = x/y
5   except TypeError:
6       print(' Type Error: Cannot divide number by string')
7   except ZeroDivisionError:
8       print('Zero Division Error: Cannot divide by zero')
9   except:
10      print('Unknown Error')
11  else:
12      print(z)
13  finally:
14      print('End of the program')
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

5.0
End of the program
```

Assertion

Testing is basically a process using which we verify and validate that an application or software or a program is free of bugs and meets all the technical requirements. There are two main ways to test your code: manually and automatically. Manually testing means you use your program and explore its features to check if everything works as expected. Automated testing involves writing scripts to test your code automatically.

Automated testing is preferable because it saves time. Instead of manually checking every feature of your code, you can write scripts to do it for you. There are two types of automated tests: unit tests and integration tests. Unit tests check small parts of your code, like individual functions, to make sure they work correctly. Integration tests check how different parts of your code work together.

A test case can be written with an assertion. An assertion is a sanity check to make sure your code isn't doing something obviously wrong. The python's built-in 'assert' keyword is used for assertion. The assert keyword lets you test if a condition in your code returns True, if not, the program will raise an AssertionError. We have to take note that assertions are not meant for mistakes made by users while they're using the program. Instead they are like alarms set up by programmers to catch their own mistakes while they're building a program. If an assertion fails during testing, it's a sign to the programmer that there's a bug that needs fixing. For users, the program should raise exceptions so that they can see a clear message about what happened.

In code, 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

Eg:

```
1   height = int(input("Enter the height of the triangle (number of rows): "))
2
3   print("Right Triangle Pattern:")
4   for i in range(1, height + 1):
5       for j in range(i):
6           print("* ", end="")
7       print()
8
9
10  assert height > 0, "Height must be a positive integer"
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

Enter the height of the triangle (number of rows): 0
Right Triangle Pattern:
Traceback (most recent call last):
  File "
    assert height > 0, "Height must be a positive integer"
AssertionError: Height must be a positive integer
```

Writing tests in this way is okay for a simple check, but what if more than one fails? This is where test runners come in. The test runner is a special application designed for running tests, checking the output, and giving you tools for debugging and diagnosing tests and programs. There are many test runners available for Python but the one built into the Python standard library is called unittest.

To convert the earlier example to a unittest test case, you would have to:

- Import unittest from the standard library
- Create a class that inherits from the TestCase class
- Convert the test functions into methods by adding self as the first argument
- Change the assertions to use the self.assertEqual() method on the TestCase class
- Change the command-line entry point to call unittest.main()

Eg:

```python
import unittest

def triangle(height):
    pattern = ''
    for i in range(1, height + 1):
        for j in range(i):
            pattern += "* "
        pattern += '\n'
    return pattern

class TestTriangle(unittest.TestCase):
    def test_triangle_positive_height(self):
        self.assertEqual(triangle(5), "* \n* * \n* * * \n* * * * \n* * * * * \n", 'Height must be a positive integer')

    def test_triangle_zero_height(self):
        self.assertEqual(triangle(0), '', 'Height must be a positive integer')

    def test_triangle_negative_height(self):
        self.assertEqual(triangle(-5), '', 'Height must be a positive integer')

unittest.main()
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

Reference

Automate the boring stuff with python (no date) Automate the Boring Stuff with Python book cover thumbnail. Retrieved from: https://automatetheboringstuff.com/2e/chapter11/.

8. errors and exceptions (no date) Python documentation. Retrieved from: https://docs.python.org/3/tutorial/errors.html.

GeeksforGeeks (2023) Python exception handling, GeeksforGeeks. Retrieved from: https://www.geeksforgeeks.org/python-exception-handling/.

Admin (2023) Differences between testing and debugging, BYJUS. Retrieved from: https://byjus.com/gate/difference-between-testing-and-debugging/#:~:text=Testing%20is%20the%20process%20using,differences%20between%20testing%20and%20debugging.

Real Python (2023) Getting started with testing in Python, Real Python. Retrieved from: https://realpython.com/python-testing/