Errors, Exceptions, and Testing

David Carlson

March 8, 2021

Types of Errors

- Syntax error
  - Errors related to language structure.

Types of Errors

- Syntax error
  - Errors related to language structure.
- Runtime error
  - Errors during the execution of program.
  - eg. TypeError, NameError

Types of Errors

- Syntax error
  - Errors related to language structure.
- Runtime error
  - Errors during the execution of program.
  - eg. TypeError, NameError
- Semantic error
  - The program will run successfully but the output is not what you expect.
  - You'll need to run a test.

Debugging Tips

Make sure:

- You are not using a reserved/keyword.

```
>>> import keyword
>>> keyword.kwlist
```

Debugging Tips

Make sure:

- You are not using a reserved/keyword.

```
>>> import keyword
>>> keyword.kwlist
```

- You have `:` after `for`, `while`, etc.

Debugging Tips

Make sure:

- You are not using a reserved/keyword.

```
>>> import keyword
>>> keyword.kwlist
```

- You have `:` after `for`, `while`, etc.
- Parentheses and quotations are closed properly.

Debugging Tips

Make sure:

- You are not using a reserved/keyword.

```
>>> import keyword
>>> keyword.kwlist
```

- You have `:` after `for`, `while`, etc.
- Parentheses and quotations are closed properly.
- You use $=$ and $==$ correctly.

Debugging Tips

Make sure:

- You are not using a reserved/keyword.

```
>>> import keyword
>>> keyword.kwlist
```

- You have `:` after `for`, `while`, etc.
- Parentheses and quotations are closed properly.
- You use $=$ and $==$ correctly.
- Indentation is correct!

Exceptions

- `raise:  #to create exceptions or errors`

Exceptions

- `raise:  #to create exceptions or errors`
- `pass #to continue execution without doing anything`

Exceptions

- `raise:  #to create exceptions or errors`
- `pass #to continue execution without doing anything`
- `try:  #tries executing the following`

Exceptions

- `raise:  #to create exceptions or errors`
- `pass #to continue execution without doing anything`
- `try:  #tries executing the following`

```
  ....
except TypeError:
  ... # runs if a Type Error was raised
except:
  ... # runs for other errors or exceptions
else:
  ... # runs if there was no exception/error
finally:
  ... # always runs!
```

Exceptions

- `raise:  #to create exceptions or errors`
- `pass #to continue execution without doing anything`
- `try:  #tries executing the following`

```
  ....
except TypeError:
  ... # runs if a Type Error was raised
except:
  ... # runs for other errors or exceptions
else:
  ... # runs if there was no exception/error
finally:
  ... # always runs!
```

- You can create your own exceptions using classes.

Unit Testing

- Write tests before or as you write code.

Unit Testing

- Write tests before or as you write code.
- Test the smallest possible *unit*.

Unit Testing

- Write tests before or as you write code.
- Test the smallest possible *unit*.
- Automate tests.

Unit Testing

- Write tests before or as you write code.
- Test the smallest possible *unit*.
- Automate tests.
- Test-driven development.

Why Unit Test?

- Find bugs quickly.

Why Unit Test?

- Find bugs quickly.
- Forces code structure.

Why Unit Test?

- Find bugs quickly.
- Forces code structure.
- Allows easier integration of multiple functions.

Why Unit Test?

- Find bugs quickly.
- Forces code structure.
- Allows easier integration of multiple functions.
- Much easier to return to code.
  - Write a test for what you want to implement next.

Why Unit Test?

- Find bugs quickly.
- Forces code structure.
- Allows easier integration of multiple functions.
- Much easier to return to code.
  - Write a test for what you want to implement next.
- Easier to make code changes.
- You can easily incorporate lots of these into your work flow.

Sample Test

```
import unittest #You need this module
import myscript #This is the script you want to test


class mytest(unittest.TestCase):

  def test_one(self):
    self.assertEqual("result I need", myscript.myfunction(myinput))

  def test_two(self)
    thing1=myscript.myfunction(myinput1)
    thing2=myscript.myfunction(myinput2)
    self.assertNotEqual(thing1, thing2)

if __name__ == '__main__': #Add this if you want to run the test with this script.
  unittest.main()
```

Test Functions

- `self.assertEqual(,)`

Test Functions

- `self.assertEqual(,)`
- `self.assertNotEqual(,)`

Test Functions

- `self.assertEqual(,)`
- `self.assertNotEqual(,)`
- `self.assertTrue()`

Test Functions

- `self.assertEqual(,)`
- `self.assertNotEqual(,)`
- `self.assertTrue()`
- `self.assertFalse()`

Test Functions

- `self.assertEqual(,)`
- `self.assertNotEqual(,)`
- `self.assertTrue()`
- `self.assertFalse()`
- `self.assertRaises()`

Test Functions

- `self.assertEqual(,)`
- `self.assertNotEqual(,)`
- `self.assertTrue()`
- `self.assertFalse()`
- `self.assertRaises()`

Useful link: `https://docs.python.org/3/library/unittest.html`

Break, Continue and Else

- These statements can be handy using `while` or `for` loops.

Break, Continue and Else

- These statements can be handy using `while` or `for` loops.
- `break #stops the loop`

Break, Continue and Else

- These statements can be handy using `while` or `for` loops.
- `break #stops the loop`
- `continue # moves on to the next iteration`

Break, Continue and Else

- These statements can be handy using `while` or `for` loops.
- `break #stops the loop`
- `continue # moves on to the next iteration`
- `else #executed only if all iterations are completed`

Break, Continue and Else

- These statements can be handy using `while` or `for` loops.
- `break #stops the loop`
- `continue # moves on to the next iteration`
- `else #executed only if all iterations are completed`

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     else:
...         print(n, 'is a prime number')
...
```