Exception Handling

Erin Rossiter

Today

1. 3 types of programming errors
2. Debugging
3. Exception handling
4. Unit tests
5. `break`, `continue`, `else`

Types of Errors

- Syntax error
  - Errors related to language structure.
  - Forgotten symbols, typos, or confusing object names.
  - Pre-runtime; parser doesn't understand; fatal
  - Check the ^!

```
In [35]: if True:
    ...:     print "hello"
    ...: else
    ...:     print "bye"
File "<ipython-input-35-c43b9c02df68>", line 3
    else
        ^
SyntaxError: invalid syntax
```

Types of Errors

- Runtime error
  - Errors during the execution of program.
  - eg. TypeError, NameError, ZeroDivisionError

Types of Errors

- Runtime error
  - Errors during the execution of program.
  - eg. TypeError, NameError, ZeroDivisionError

```
>>> callMe = "Maybe"
>>> print(callme)
Traceback (most recent call last):
  In line 2 of the code you submitted:
    print(callme)
NameError: name 'callme' is not defined
```

Types of Errors

- Runtime error
  - Errors during the execution of program.
  - eg. TypeError, NameError, ZeroDivisionError

```
>>> callMe = "Maybe"
>>> print(callme)
Traceback (most recent call last):
  In line 2 of the code you submitted:
    print(callme)
NameError: name 'callme' is not defined
>>> print("you cannot add text and numbers" + 12)
Traceback (most recent call last):
  In line 1 of the code you submitted:
    print("you cannot add text and numbers" + 12)
TypeError: Can't convert 'int' object to str implicitly
```

Types of Errors

- Semantic error
  - The program will run successfully but the output is not what you expect.

Types of Errors

- Semantic error
  - The program will run successfully but the output is not what you expect.
  - Task: create a program that calculates the average of two numbers ($\frac{x+y}{2}$)

```
>>>  x = 3
>>>  y = 4
>>> average = x + y / 2
>>> print(average)

5.0 # ????
```

Types of Errors

- Semantic error
  - The program will run successfully but the output is not what you expect.
  - Task: create a program that calculates the average of two numbers ($\frac{x+y}{2}$)

```
>>>  x = 3
>>>  y = 4
>>> average = x + y / 2
>>> print(average)

5.0 # ????
```

  - Very common, very annoying and, unfortunately, without indication that they exist.

Types of Errors

- Semantic error
  - The program will run successfully but the output is not what you expect.
  - Task: create a program that calculates the average of two numbers ($\frac{x+y}{2}$)

```
>>>  x = 3
>>>  y = 4
>>> average = x + y / 2
>>> print(average)

5.0 # ????
```

  - Very common, very annoying and, unfortunately, without indication that they exist.
  - So we debug and test.

Types of Errors: Review

A *syntax error* happens when Python can't understand what you are saying.

```
x =* 2
```

A *run-time error* happens when Python understands what you are saying, but runs into trouble when following your instructions.

```
12 + "hi"
```

A *semantic* errors happens when Python understands what you are saying and can do it, but you wanted something else.

`x = y` vs. `x == y`

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`, `if`, `else`, `def`, etc.

Debugging Tips

Make sure:

- You are not using a reserved/keyword.

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

- You have `:` after `for`, `while`, `if`, `else`, `def`, 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`, `if`, `else`, `def`, 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`, `if`, `else`, `def`, etc.
- Parentheses and quotations are closed properly.
- You use = and == correctly.
- Indentation is correct! Remember, even spaces in empty lines count.

Debugging Tips

Make sure:

- You are not using a reserved/keyword.

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

- You have `:` after `for`, `while`, `if`, `else`, `def`, etc.
- Parentheses and quotations are closed properly.
- You use = and == correctly.
- Indentation is correct! Remember, even spaces in empty lines count.
- Remember python starts indexing at 0!

Exceptions

- Use when we expect an error might occur.

Exceptions

- Use when we expect an error might occur.
- Write code that runs only under those circumstances to handle the error.

Exceptions

- Use when we expect an error might occur.
- Write code that runs only under those circumstances to handle the error.
- You might expect multiple kinds of errors, handle each differently.

Exceptions

- Use when we expect an error might occur.
- Write code that runs only under those circumstances to handle the error.
- You might expect multiple kinds of errors, handle each differently.
- Typical structure:

```
try:
  ....# tries to executing the following
except TypeError:
  ... # runs if a Type Error was raised
except AttributeError:
  ... # runs for other errors or exceptions
else:
  ... # runs if there was no exception/error
finally:
  ... # always runs!
```

Exceptions

- You can create your own exceptions using classes.
  - You can `raise` them when something goes wrong

Exceptions

- You can create your own exceptions using classes.
  - You can `raise` them when something goes wrong
- We will go over lots of examples.

Exceptions

- You can create your own exceptions using classes.
  - You can `raise` them when something goes wrong
- We will go over lots of examples.
- I use built-in exceptions a lot in my coding.

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, especially semantic errors.

Why Unit Test?

- Find bugs quickly, especially semantic errors.
- Forces code structure.

Why Unit Test?

- Find bugs quickly, especially semantic errors.
- Forces code structure.
- Allows easier integration of multiple functions.

Why Unit Test?

- Find bugs quickly, especially semantic errors.
- Forces code structure.
- Allows easier integration of multiple functions.
- Much easier to return to code.
  - Advice is to write a test for what you want to implement next.

Why Unit Test?

- Find bugs quickly, especially semantic errors.
- Forces code structure.
- Allows easier integration of multiple functions.
- Much easier to return to code.
  - Advice is to 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/2/library/unittest.html`

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

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')
...
```