CME 193: Introduction to Scientific Python

Lecture 8: Exceptions, testing and more modules

Sven Schmit

`stanford.edu/~schmit/cme193`

Contents

Wordcount

Recall the wordcount exercise:

Write a function that takes a filename, and returns the 20 most common words.

Suppose we have written a function `topkwords(filename, k)`

Instead of entering filename and value of k in the script, we can also run it from the terminal.

Parse input from command line

The sys module allows us to read the terminal command that started the script:

```
import sys

print sys.argv
```

sys.argv holds a list with command line arguments passed to a Python script.

Parse input from command line

The sys module allows us to read the terminal command that started the script:

```
import sys

print sys.argv
```

sys.argv holds a list with command line arguments passed to a Python script.

Back to the wordcount example

```
import sys

def topkwords(filename, k):
    # Returns k most common words in filename
    pass

if __name__ == "__main__":
    filename = sys.argv[1]
    k = int(sys.argv[2])
    print topkwords(filename, k)
```

Issues?

Back to the wordcount example

```
import sys

def topkwords(filename, k):
    # Returns k most common words in filename
    pass

if __name__ == "__main__":
    filename = sys.argv[1]
    k = int(sys.argv[2])
    print topkwords(filename, k)
```

Issues?

Issues

- What if the file does not exist?
- What if the second argument is not an integer?
- What if no command line arguments are supplied?

All result in errors:

- IOError
- ValueError
- IndexError

Issues

- What if the file does not exist?
- What if the second argument is not an integer?
- What if no command line arguments are supplied?

All result in errors:

- IOError
- ValueError
- IndexError

Issues

- What if the file does not exist?
- What if the second argument is not an integer?
- What if no command line arguments are supplied?

All result in errors:

- IOError
- ValueError
- IndexError

Exception handling

What do we want to happen when these errors occur? Should the program simply crash?

No, we want it to gracefully handle these

- IOError: Tell the user the file does not exist.
- ValueError, IndexError: Tell the user what the format of the command line arguments should be.

Exception handling

What do we want to happen when these errors occur? Should the program simply crash?

No, we want it to gracefully handle these

- IOError: Tell the user the file does not exist.
- ValueError, IndexError: Tell the user what the format of the command line arguments should be.

Try ... Except

```
import sys

if __name__ == "__main__":
    try:
        filename = sys.argv[1]
        k = int(sys.argv[2])
        print topkwords(filename, k)
    except IOError:
        print "File does not exist"
    except (ValueError, IndexError):
        print "Error in command line input"
        print "Run as: python wc.py <filename> <k>"
        print "where <k> is an integer"
```

Try ... Except

- The try clause is executed
- If no exception occurs, the except clause is skipped
- If an exception occurs, the rest of the try clause is skipped. Then if the exception type is matched, the except clause is executed. Then the code continues after the try statement
- If an exception occurs with no match in the except clause, execution is stopped and we get the standard error

Try ... Except

- The try clause is executed
- If no exception occurs, the except clause is skipped
- If an exception occurs, the rest of the try clause is skipped. Then if the exception type is matched, the except clause is executed. Then the code continues after the try statement
- If an exception occurs with no match in the except clause, execution is stopped and we get the standard error

Try ... Except

- The try clause is executed
- If no exception occurs, the except clause is skipped
- If an exception occurs, the rest of the try clause is skipped. Then if the exception type is matched, the except clause is executed. Then the code continues after the try statement
- If an exception occurs with no match in the except clause, execution is stopped and we get the standard error

A naked except

We can have a naked except that catches any error:

```
try:
    t = 3.0 / 0.0
except:
    # handles any error
    print 'There was some error'
```

Use this with extreme caution though, as genuine bugs might be impossible to correct!

Try Except Else

The else clause after all except statements is executed after succeful execution of the try block (hence, when no exception was raised)

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'cannot open', arg
    else:
        print arg, 'has', len(f.readlines()), 'lines'
        f.close()
# from Python docs
```

Raise

We can use `Raise` to raise an exception ourselves.

```
>>> raise NameError('Oops')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: Oops
```

Finally

The `finally` statement is always executed before leaving the `try` statement, whether or not an exception has occured.

```
def div(x, y):
    try:
        return x/y
    except ZeroDivisionError:
        print 'Division by zero!'
    finally:
        print "Finally clause"

print div(3,2)
print div(3,0)
```

Useful in case we have to close files, closing network connections etc.

Finally

The `finally` statement is always executed before leaving the `try` statement, whether or not an exception has occured.

```
def div(x, y):
    try:
        return x/y
    except ZeroDivisionError:
        print 'Division by zero!'
    finally:
        print "Finally clause"

print div(3,2)
print div(3,0)
```

Useful in case we have to close files, closing network connections etc.

Contents

Unit testing

Unit tests are scripts that test whether your code runs as intended.

For example, when writing a factorial function, we can check

- $0! = 1$
- $3! = 6$
- etc.

Test driven development

Some find unit testing so important, they write unit tests before they write the code they want to test.

Reasons:

- Focus on the requirements you need
- You do not write too much code: once unit tests pass, you are done
- When refactoring code, you know all still works as the old version
- When working together, you know you aren't breaking other people's code
- It's faster, and you can be confident in your code.

Test cases

The fundamental part of unit testing is constructing the tests.

A test case should answer a single question about the code, it should

- Run by itself, no human input required
- Determine on its own whether the test has passed or failed
- Be seperate from other tests

unittest

The standard Python module `unittest` helps you write unit tests.

```
import unittest
from my_script import is_palindrome

class KnownInput(unittest.TestCase):
    knownValues = (('lego', False),
            ('radar', True))

    def testKnownValues(self):
        for word, palin in self.knownValues:
            result = is_palindrome(word)
            self.assertEqual(result, palin)
```

Nose is a module that makes testing even easier, especially if you have a project with a lot of test scripts.

What to test

- Known values
- Sanity check (for conversion functions for example)
- Bad input
  - Input is too large?
  - Negative input?
  - String input when expected an integer?
- etc: very dependent on problem

assert

We can use a number of methods to check for failures:

- `assertEqual`
- `assertNotEqual`
- `assertTrue`, `assertFalse`
- `assertIn`
- `assertRaises`
- `assertAlmostEqual`
- `assertGreater`, `assertLessEqual`
- etc. (see Docs)

Some tips for exercise 15.2

Problem: Write a program `primes.py` that takes two command line arguments, $a < b$, and returns all prime numbers between a and b (inclusive). Write a seperate test script `primestest.py` that contains unittests for all functions in your script.

Functions:

1. Extract a and b from command line arguments
2. Function that tests whether a number is prime: `is_prime`
3. Function that uses the `is_prime` function to construct a list with prime numbers between a and b

Then your script can combine all of these

Some tips for exercise 15.2

Problem: Write a program `primes.py` that takes two command line arguments, $a < b$, and returns all prime numbers between a and b (inclusive). Write a seperate test script `primestest.py` that contains unittests for all functions in your script.

Functions:

1. Extract a and b from command line arguments
2. Function that tests whether a number is prime: `is_prime`
3. Function that uses the `is_prime` function to construct a list with prime numbers between a and b

Then your script can combine all of these

tests for is_prime

Tests:

1. 0 is not prime
2. 1 is not prime
3. 2 is prime
4. 3 is prime
5. 4 is not prime
6. -3 is not prime
7. 9 is not prime
8. 13 is prime
9. 31 is prime

tests for is_prime

Set up functions

```
def is_prime(x):
    pass

def primes_between(a, b):
    pass

def read_ab_cmd_line(argv):
    pass
```

(Some prefer to do this step after writing tests)

tests for is_prime

Write tests...

```
import unittest
import unit_primes as primes

class TestIsPrime(unittest.TestCase):
    def test_zero(self):
        self.assertFalse(primes.is_prime(0))

    def test_two(self):
        self.assertTrue(primes.is_prime(2))

    def testPrimes(self):
        prime_numbers = [7, 11, 13, 31]
        for prime in prime_numbers:
            self.assertTrue(primes.is_prime(prime))

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

then update functions until all tests pass.

Contents

More modules

In the next few slides, some libraries are introduced that might be useful for specific tasks.

They are meant to show you what possibilities are out there.

Pickle

Module for *serializing* and *deserializing* objects in Python.

Can be used to write any Python object to a file (`dump`) and load it again later (`load`).

Very very simple and extremely useful.

`cPickle` is similar but implemented in C under the hood, and hence can be significantly faster.

Pickle

Module for *serializing* and *deserializing* objects in Python.

Can be used to write any Python object to a file (`dump`) and load it again later (`load`).

Very very simple and extremely useful.

`cPickle` is similar but implemented in C under the hood, and hence can be significantly faster.

Regular expressions

A *regular expression* (RE) specify a set of strings that matches it. This module can be used to check whether a particular string matches the RE

see also:

`https://docs.python.org/2/howto/regex.html#regex-howto`

Regular expressions

`. ^ $ * + ? { } [ ] \ ( )|`

[and]	are used for specifying a *character class*, e.g.[aeiou], [A-Z], [A-z], [0-5]
$\wedge$	is the complement character, e.g. $[\wedge 5]$ matches all except for a '5'.
$\backslash$	used to signal various special sequences, including the use of metacharacters, e.g. $\backslash\wedge$ to match $\wedge$.
characters	usually map to characters: 'test' - 'test'
\d	matches any decimal digit: '\d' - '[0-9]'

Regular expressions

Suppose you want to find phone numbers:

- 1234567890
- 123-456-7890
- (123) 465 7890
- (123) 456-7890
- etc

How to find all these?

```
p = re.compile('\(?(\d{3})\[-\)]?\s?(\d{3})[-\s]?(\d{4})$')
```

Regular expressions

Suppose you want to find phone numbers:

- 1234567890
- 123-456-7890
- (123) 465 7890
- (123) 456-7890
- etc

How to find all these?

```
p = re.compile('\(?(\d{3})\[-\)]?\s?(\d{3})[-\s]?(\d{4})$')
```

Regular expressions

Pattern:

- Maybe a bracket: \(?
- 3 numbers: \d{3}
- Maybe a bracket or a hyphen: [-\)]?
- Maybe a whitespace: \s?
- 3 numbers: \d{3}
- Maybe a hyphen or a whitespace: [-\s]?
- 4 numbers: \d{4}
- End: $

Extract the numbers by placing brackets: (\d{3}) around numbers

```
'\(?(\d{3})[-\)]?\s?(\d{3})[-\s]?(\d{4})$'
```

Regular expressions

Pattern:

- Maybe a bracket: \(?
- 3 numbers: \d{3}
- Maybe a bracket or a hyphen: [-\)]?
- Maybe a whitespace: \s?
- 3 numbers: \d{3}
- Maybe a hyphen or a whitespace: [-\s]?
- 4 numbers: \d{4}
- End: $

Extract the numbers by placing brackets: (\d{3}) around numbers

```
'\(?(\d{3})[-\)]?\s?(\d{3})[-\s]?(\d{4})$'
```

Regular expressions

```
import re

pat = '\(?(\d{3})[-\)]?\s?(\d{3})[-\s]?(\d{4})$'
repat = re.compile(pat)
string = '(123) 456-7890'
search =  repat.search(string)
if search:
    print search.groups()
else:
    print 'not found'
```

How to test?

Unit testing is invented for these kind of problems!

Regular expressions

```
import re

pat = '\(?(\d{3})[-\)]?\s?(\d{3})[-\s]?(\d{4})$'
repat = re.compile(pat)
string = '(123) 456-7890'
search =  repat.search(string)
if search:
    print search.groups()
else:
    print 'not found'
```

How to test?

Unit testing is invented for these kind of problems!

Regular expressions

```
import re

pat = '\(?(\d{3})[-\)]?\s?(\d{3})[-\s]?(\d{4})$'
repat = re.compile(pat)
string = '(123) 456-7890'
search =  repat.search(string)
if search:
    print search.groups()
else:
    print 'not found'
```

How to test?

Unit testing is invented for these kind of problems!

Requests

HTTP library for Python.

Alternative: urllib, urllib2

Beautiful soup

"You didn't write that awful page. You're just trying to get some data out of it. Beautiful Soup is here to help. Since 2004, it's been saving programmers hours or days of work on quick-turnaround screen scraping projects."

Useful for scraping HTML pages.

Such as: finding all links, or specific urls.

Get data from poorly designed websites.

Scikits

Additional packages that extend Scipy:

- skikit-aero
- scikit-learn
- scikit-image
- cuda
- odes

Scikit learn

Large Scikit package with a lot of functionality. Sponsored by INRIA (and Google sometimes)

- Classification
- Regression
- Clustering
- Dimensionality reduction
- Model selection
- Preprocessing

Pandas

Python Data Analysis library.

High performance data structures and data analysis tools.

See the Pandas Cookbook if interested for a great tutorial showing off much of the functionality:
`http://pandas.pydata.org/pandas-docs/stable/tutorials.html`

Contents

Project and portfolio

Please remember: projects and portfolios due next Monday at noon.

Coursework checklist:

- Project code (**py** file(s) or **zip**).
- Project write-up (**pdf** file (no word!)).
- All scripts you wrote for class (**zip** with all **py** files). No write up necessary.

Feedback

Thank you very much for your participation. I hope you enjoyed the class and feel like you learnt a lot about Python!

I would greatly appreciate your feedback to help improve this course. Please take a minute to fill a very short feedback form here:

`http://goo.gl/forms/yhWByFl5Zx`

or via course website

Now some time to work on exercises or project, or to ask me questions!