Introduction to Python
Unicode, Advanced Argument passing
List and Dict Comprehensions, Testing

Christopher Barker

UW Continuing Education

October 29, 2013

Table of Contents

Review of Previous Class

- Dictionaries
- Exceptions
- Files, etc.

Lightning Talks

Lightning talks today:

Rithy Chhen

Howard Edson

Dong Kang

Steven Werner

Homework review

Homework Questions?

My Solution

Unicode

I hope you all read this:

The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)

`http://www.joelonsoftware.com/articles/Unicode.html`

If not – go read it!

Unicode

Everything is bytes

If it's on disk or transmitted over a network, it's bytes

Python provides some abstractions to make it easier to deal with bytes

Unicode

Unicode is a biggie

Strings vs Unicode
(`str()` vs. `bytes()` vs. `unicode()`)

Python 2.x vs 3.x

(actually, dealing with numbers rather than bytes is big – but we take that for granted)

Unicode

Strings are sequences of bytes

Unicode strings are sequences of platonic characters

Platonic characters cannot be written to disk or network!

(ANSI – one character == one byte – so easy!)

Unicode

The `unicode` object lets you work with characters

"Encoding" is converting from a unicode object to bytes

"Decoding" is converting from bytes to a unicode object

Unicode

```
import codecs
ord()
chr()
unichr()
str()
unicode()
encode()
decode()
```

Unicode Literals

1) Use unicode in your source files:

```
# -*- coding: utf-8 -*-
```

2) escape the unicode characters

```
print u"The integral sign: \u222B"
print u"The integral sign: \N{integral}"
```

lots of tables of code points online:
`http://inamidst.com/stuff/unidata/`

(demo: `code\hello_unicode.py`)

Unicode

Use unicode objects in all your code

decode on input

encode on output

Many packages do this for you
(XML processing, databases, ...)

Gotcha:
Python has a default encoding (usually ascii)

Unicode

Python Docs Unicode HowTo:

`http://docs.python.org/howto/unicode.html`

"Reading Unicode from a file is therefore simple:"

```
import codecs
f = codecs.open('unicode.rst', encoding='utf-8')
for line in f:
    print repr(line)
```

Encodings Built-in to Python:

`http://docs.python.org/2/library/codecs.html#standard-encodings`

Unicode LAB

- Find some nifty non-ascii characters you might use.
  Create a unicode object with them in two different ways.
- In the "code" dir for this week, there are two files:
  `text.utf16`
  `text.utf32`
  read the contents into unicode objects
- write some of the text from the first exercise to file.
- read that file back in.

(reference: `http://inamidst.com/stuff/unidata/`)

NOTE: if your terminal does not support unicode – you'll get an error trying to print. Try a different terminal or IDE, or google for a solution

Lightning Talk

Lightning Talks:

Rithy Chhen

Howard Edson

Keyword arguments

When defining a function, you can specify only what you need – any order

```
In [151]: def fun(x,y=0,z=0):
              print x,y,z
          .....:

In [152]: fun(1,2,3)
1 2 3

In [153]: fun(1, z=3)
1 0 3

In [154]: fun(1, z=3, y=2)
1 2 3
```

Keyword arguments

A Common Idiom:

```
def fun(x, y=None):
    if y is None:
        do_something_different

    go_on_here
```

Keyword arguments

Can set defaults to variables

```
In [156]: y = 4

In [157]: def fun(x=y):
     print "x is:", x
    .....:

In [158]: fun()
x is: 4
```

Keyword arguments

Defaults are evaluated when the function is defined

```
In [156]: y = 4

In [157]: def fun(x=y):
    ....:     print "x is:", x
    ....:

In [158]: fun()
x is: 4

In [159]: y = 6

In [160]: fun()
x is: 4
```

Function arguments in variables

function arguments are really just
– a tuple (positional arguments)
– a dict (keyword arguments)

```
def f(x, y, w=0, h=0):
    print "position: %s, %s -- shape: %s, %s"%(x, y, w, h)

position = (3,4)
size = {'h': 10, 'w': 20}

>>> f( *position, **size)
position: 3, 4 -- shape: 20, 10
```

Function parameters in variables

You can also pull in the parameters out in the function as a tuple and a dict

```
def f(*args, **kwargs):
    print "the positional arguments are:", args
    print "the keyword arguments are:", kwargs

In [389]: f(2, 3, this=5, that=7)
the positional arguments are: (2, 3)
the keyword arguments are: {'this': 5, 'that': 7}
```

LAB

keyword arguments

- Write a function that has four optional parameters (with defaults):
  - foreground_color
  - background_color
  - link_color
  - visited_link_color
- Have it print the colors.
- Call it with a couple different parameters set
- Have it pull the parameters out with `*args, **kwargs`

List comprehensions

A bit of functional programming:

```
new_list = [expression for variable in a_list]
```

same as for loop:

```
new_list = []
for variable in a_list:
    new_list.append(expression)
```

List comprehensions

More than one "for":

```
new_list = \
[exp for var in a_list for var2 in a_list2]
```

same as nested for loop:

```
new_list = []
for var in a_list:
    for var2 in a_list2:
        new_list.append(expression)
```

You get the "outer product", i.e. all combinations.

(demo)

List comprehensions

Add a conditional:

```
new_list = \
[expression for variable in a_list if something_is_true]
```

same as for loop:

```
new_list = []
for variable in a_list:
    if something_is_true:
        new_list.append(expression)
```

(demo)

List comprehensions

Examples:

```
In [341]: [x**2 for x in range(3)]
Out[341]: [0, 1, 4]

In [342]: [x+y for x in range(3) for y in range(5,7)]
Out[342]: [5, 6, 6, 7, 7, 8]

In [343]: [x*2 for x in range(6) if not x%2]
Out[343]: [0, 4, 8]
```

List comprehensions

Remember this from last week?

```
[name for name in dir(__builtin__) if "Error" in name]
```

```
['ArithmeticError',
 'AssertionError',
 'AttributeError',
 'BufferError',
 'EOFError',
 ....
```

Set Comprehensions

You can do it with sets, too:

```
new_set = { value for variable in a_sequence}
```

same as for loop:

```
new_set = set()
for key in a_list:
    new_set.add(value)
```

Set Comprehensions

```
In [33]: s = "a fairly long string"

In [34]: vowels = 'aeiou'

In [35]: { l for l in s if l in vowels}
Out[35]: set(['a', 'i', 'o'])
```

Dict Comprehensions

and with dicts:

```
new_dict = { key:value for variable in a_sequence}
```

same as for loop:

```
new_dict = {}
for key in a_list:
    new_dict[key] = value
```

Dict Comprehensions

Example

```
In [340]: { i: "this_%i"%i for i in range(5) }
Out[340]: {0: 'this_0', 1: 'this_1', 2: 'this_2',
           3: 'this_3', 4: 'this_4'}
```

(not as useful with the `dict()` constructor...)

LAB

List and Dict comprehension lab:

`code/comprehensions.rst[html]`

Lightning Talk

Lightning Talks:

Dong Kang

Steven Werner

Unit Testing

Gaining Traction

You need to test your code somehow when you write it – why not preserve those tests?

And allow you to auto-run them later?

Test-Driven development:

- Write the tests before the code

Unit Testing

My thoughts:

Unit testing encourages clean, decoupled design

If it's hard to write unit tests for – it's not well designed

but...

"complete" test coverage is a fantasy

PyUnit

PyUnit: the stdlib unit testing framework

```
import unittest
```

More or less a port of Junit from Java

A bit verbose: you have to write classes & methods

(And we haven't covered that yet!)

unittest example

```
import random
import unittest

class TestSequenceFunctions(unittest.TestCase):

    def setUp(self):
        self.seq = range(10)

    def test_shuffle(self):
        # make sure the shuffled sequence does not lose any elem
        random.shuffle(self.seq)
        self.seq.sort()
        self.assertEqual(self.seq, range(10))

        # should raise an exception for an immutable sequence
        self.assertRaises(TypeError, random.shuffle, (1,2,3))
```

unittest example (cont)

```
    def test_choice(self):
        element = random.choice(self.seq)
        self.assertTrue(element in self.seq)

    def test_sample(self):
        with self.assertRaises(ValueError):
            random.sample(self.seq, 20)
        for element in random.sample(self.seq, 5):
            self.assertTrue(element in self.seq)

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

(code/unitest_example.py)

http://docs.python.org/library/unittest.html

unittest

Lots of good tutorials out there:

Google: "python unittest tutorial"

I first learned from this one:

`http://www.diveintopython.net/unit_testing/index.html`

nose and pytest

Due to its Java heritage, unittest is kind of verbose

Also no test discovery
(though unittest2 does add that...)

So folks invented nose and pytest

nose

nose

Is nicer testing for python

nose extends unittest to make testing easier.

```
$ pip install nose

$ nosetests unittest_example.py
```

http://nose.readthedocs.org/en/latest/

nose example

The same example – with nose

```
import random
import nose.tools

seq = range(10)

def test_shuffle():
    # make sure the shuffled sequence does not lose any elements
    random.shuffle(seq)
    seq.sort()
    assert seq == range(10)

@nose.tools.raises(TypeError)
def test_shuffle_immutable():
    # should raise an exception for an immutable sequence
    random.shuffle( (1,2,3) )
```

nose example (cont)

```
def test_choice():
    element = random.choice(seq)
    assert (element in seq)

def test_sample():
    for element in random.sample(seq, 5):
        assert element in seq

@nose.tools.raises(ValueError)
def test_sample_too_large():
    random.sample(seq, 20)
```

(code/test_random_nose.py)

pytest

pytest

- A mature full-featured testing tool
- Provides no-boilerplate testing
- Integrates many common testing methods

```
$ pip install pytest

$ py.test unittest_example.py
```

http://pytest.org/latest/

pytest example

The same example – with pytest

```
import random
import pytest

seq = range(10)

def test_shuffle():
    # make sure the shuffled sequence does not lose any elements
    random.shuffle(seq)
    seq.sort()
    assert seq == range(10)

def test_shuffle_immutable():
    pytest.raises(TypeError, random.shuffle,  (1,2,3) )
```

pytest example (cont)

```
def test_choice():
    element = random.choice(seq)
    assert (element in seq)

def test_sample():
    for element in random.sample(seq, 5):
        assert element in seq

def test_sample_too_large():
    with pytest.raises(ValueError):
        random.sample(seq, 20)
```

(code/test_random_pytest.py)

Parameterized Tests

A whole set of inputs and outputs to test?
pytest has a nice way to do that (so does nose...)

```
import pytest
@pytest.mark.parametrize(("input", "expected"), [
    ("3+5", 8),
    ("2+4", 6),
    ("6*9", 42),
])
def test_eval(input, expected):
    assert eval(input) == expected
```

http://pytest.org/latest/example/parametrize.html

(code/test_pytest_parameter.py)

Test Coverage

`coverage.py`

Uses debugging hook to see which lines of code are actually executed – plugins exist for most (all?) test runners

```
pip install coverage
```

```
nosetests --with-coverage test_codingbat.py
```

`http://nedbatchelder.com/code/coverage/`

Coding Bat

Coding Bat:

`http://codingbat.com/python`

Tells you what unit tests to write:

`http://codingbat.com/prob/p118406`

We'll use them for our lab

LAB

First: get pip installed:

`http://www.pip-installer.org/en/latest/installing.html`

Second: install nose and/or pytest:

`pip install nose` – `pip install pytest`

Unit Testing:

- pytest / nose
  - Test a `codingbat.com` with nose or pytest
  - Try doing test-driven development (`code\test_codingbat.py`)
- try running `coverage` on your tests

Homework

Recommended Reading:

- TP: ch 15-18
- LPTHW: Ex 40 - 45
- Dive Into Python: chapter 4, 5

Do:

- Finish (or re-factor) the Labs you didn't finish in class.
- Write some unit tests for a couple of the functions you've written for previous excercises (Or something new)
- Using the unit tests you jsut wrote, refactor the above functions using list and/or dict comprehensions.
- Write a script which does something useful (to you) and reads and writes files. Very, very small scope is good. something useful at work would be great, but no job secrets!
- Start thinking about what you want to do for your project!