CSC108H Lecture 8

Dan Zingaro
OISE/UT

July 2, 2009

Clickers: Lists and Dictionaries

Which of the following **is** a difference between lists and dictionaries?

- A. List items cannot be mutable, but dictionary values can be mutable
- B. Assigning to an offset that does not exist in a list is an error, but assigning to a key that does not exist in a dictionary is not
- C. A list can contain a dictionary as one of its items, but a dictionary cannot contain a list as one of its values
- D. There is a `dict` constructor that creates a dictionary from a suitable object, but there is no `list` constructor that similarly creates lists

Two Abs Functions (two_abs.py)

Both functions below are buggy. How can you show this?

```
def abs1(val):
  '''Return absolute value of numeric argument.'''

  if val > 0:
    return val
  if val < 0:
    return -val

def abs2(val):
  '''Return absolute value of numeric argument.'''

  if val <= -1:
    return -val
  else:
    return val
```

Manually Verifying

- Throughout the course we've been following the approach: design, code, verify
- To "verify", we've been calling our functions with particular inputs and looking at the value returned to see whether it matches what we expect
- In the above `abs` examples, what are the test cases that would help convince us of correctness?
- Each time we modify the code, we must re-verify (why?)
- We're now going to save our verification work, so that we can reuse it when the code changes
- We will use a module called nose, which is useful for writing test cases

Verifying with Nose

- The procedure for testing with Nose is as follows
  - Create a new module: typically named `test_mod`, where `mod` is replaced with the name of the module we are testing
  - `import nose`
  - `import` the module to test
  - In the `if __name__` section of the module call `nose.runmodule()`

Verifying with Nose...

- The module will contain one or more functions named `test_func`, where `func` is replaced with a description of what is being tested
- Each test function will contain one `assert` statement
  - `assert value1 == value2, description`
  - `value1` is typically a call to the function with particular inputs
  - `value2` is the value that we expect the function to return
  - `description` is a string describing the test case

Testing an Abs Function (test_two_abs.py)

Here's how we might test abs1. Let's add test cases for abs2.

```
import two_abs
import nose

def test_abs1_positive():
  assert two_abs.abs1 (4) == 4, \
         'Positive number'

def test_abs1_zero():
  assert two_abs.abs1 (0) == 0, \
         'Zero'

def test_abs1_negative():
  assert two_abs.abs1 (-4) == 4, \
         'Negative number'


if __name__ == "__main__":
  nose.runmodule()
```

Results of Testing

- The first line of the output from Nose tells us which tests passed or failed
- A dot means “pass”; an F means “fail”
- For each failed test, we get information about which assertion was violated
- This information is helpful when we try to fix the errors in our code
- The last portion of output tells us the number of tests executed and the number of failures

Two More Functions

Given just the docstrings, we can still generate some test cases.
What are the benefits of not looking at the code itself?
Drawbacks?

```
def our_max(num1, num2):
  '''Return the larger of the numbers num1 and num2.'''

def are_same_string(str1, str2):
  '''Return True if strings str1 and str2 are the same
  strings ignoring case, and return False otherwise.'''
```

Testing the Two Functions (test_two_funcs.py)

Here are a couple of test cases. Let's add more to improve coverage.

```
import nose
import two_funcs

def test_our_max_first_bigger():
  assert two_funcs.our_max(8, 6) == 8, \
         'First number is larger.'

def test_are_same_string_empty():
  assert two_funcs.are_same_string("", ""), \
         'The empty string.'


if __name__ == '__main__':
    nose.runmodule()
```

A List Function (list_func.py)

```
def uc_list (lst):
  '''Return a list that contains the strings from list lst
  but with uppercase letters. lst is a list of strings.'''

  res = []
  for s in lst:
    res.append (s.upper())
```

Let's write test cases: empty list, list with one string, list with multiple strings, different cased letters, empty string ...

A Dictionary Function (dict_func.py)

```
def inc_count(d, k):
  '''Increment the value associated with key k in dict d.
  If k is not a key in d, add it with value 1.'''

  if d.has_key(k):
    d[k] += 1
  else:
    d[k] = 1
```

Let's write test cases. What should they be?

A Picture Function (pic_func.py)

```
import media

def swap_blue(pic1, pic2):
  '''Swap the blue color components of all the pixels
  in Picture pic1 with the blue color values of the
  corresponding pixels in Picture pic2. Assume
  that pic1 and pic2 have the same width and height.'''

  pic1_pixels = media.get_pixels(pic1)
  pic2_pixels = media.get_pixels(pic2)

  for i in range(len(pic1_pixels)):
    temp = media.get_blue(pic1_pixels[i])
    media.set_blue(pic1_pixels[i], \
                   pic2_pixels[i].get_blue())
    media.set_blue(pic2_pixels[i], temp)
```

A Picture Function...

- Nose can't open a picture and look at it to tell us if the function works properly
- What we will do is create two small (2x2) pictures
- Initially, `pic1` will be blue and `pic2` will be black
- Then, we call `swap_blue`
- It has worked correctly if each pixel in `pic1` is black, and each pixel in `pic2` is blue
- Given a function `make_col(pic, col)` that makes each pixel of a picture the color `col`, let's write the test case ...

Testing the Picture Function (test_swap_blue.py)

```
import media
import nose
import pic_func

def make_col (pic, col):
  '''Set each pixel of pic to color col.'''

  for pix in pic:
    media.set_color (pix, col)

def test_swap_blue():
  '''Test swap_blue using two 2 x 2 pictures, one black
  and one blue. After swap_blue has been called on the
  two pictures, the black picture should have turned
  blue, and the blue picture should have turned black.'''
```

Speed of Execution

- Python is an interpreted language
- This means that each line that is about to be executed has to be translated to machine code first
- This is usually slower than fully compiled languages like C; C programs have no "translation step"
- Java takes a middle-ground approach: it compiles code to machine language as it interprets (called just-in-time compiling)
- For Python, we have the just-in-time compiler Psyco
- Psyco offers typical speedups of 2X to 4X
- `http://psyco.sourceforge.net/`

Psyco Example

- We will give a program that wastes a lot of time computing results that are just thrown away
- If we `import time`, we can use `time.clock()` to get a time value in seconds
- To time an execution, we can compare the return value of `time.clock()` before and after the code runs

Psyco Example... (slow_code.py)

- This code is optimized using `psyco.full()`
- It is 6X faster using Psyco on my laptop

```
import time
import math
import psyco
psyco.full()

def waste():
  for i in range(3000):
    x = math.sin(i) * math.pi * i

start = time.clock()
for i in range(1000):
  res = waste()
finish = time.clock()
elapsed = finish - start
print elapsed
```

Packaging Applications

- So, Python is interpreted. How can we give our software to people that don't have Python installed?
- Various packaging tools: Py2exe (Windows), PyInstaller (Windows, Linux)
- They package your source code with Python libraries that interpret your code
- Let's see what Py2exe does!

Review Questions

List Occurrences

Let's write the function for the following header and docstring.

```
def count_occurrences(L):
  '''Given a list, return a dictionary in which the keys
  are the items in the list and their associated values
  are integers denoting the number of times the item is
  contained in the list.'''
```

Folding Dictionaries

Let's write the function for the following header and docstring. Is the resulting dictionary **guaranteed** to be unique?

```
def fold(d1, d2):
  '''Given two dictionaries, d1 and d2, return a new
  dictionary that contains all (b, c) such that
  (a, b) is in d1 and (a, c) is in d2.'''
```

Sum of List Values

Let's write the function for the following header and docstring.

```
def combine1(d1, d2):
  '''Return the dictionary that contains all keys that are
  keys in both d1 and d2.  The value associated with each
  key in the new dictionary is the sum of all the integers
  associated with that key in d1 and d2.  The values in
  d1 and d2 are lists of integers.'''
```