CIS 480 - Python - Fall 2005
Homework #6

(note: there is NO Week 8 Lab Exercise, as there is no Week 8 Lab...)

HW #6 due: Thursday, October 20th, 12:00 noon

Until I develop more formal opening comment block standards (which I STILL plan to do), begin **each python module** that you write with at least the following opening comments:

- a comment containing the name of the module (the **file name** of the module, please --- **hw07.py**, for example)
- a comment containing your name, and
- a comment containing the date that your module was last modified

HOMEWORK #6

Create a Python module **hw06.py**. Within it, include the following:

1. Since we talked about **types** in Python this week, writing some predicate functions that let us know if arguments are certain types seems appropriate. (For our purposes, a predicate function is one that returns the **bool** value **True** or the **bool** value **False**.)

First: write a predicate function **single_letter_key_dict**. It takes any single argument, and returns the **bool** value **True** if that argument is a dict whose keys are **all** single-letter strings, and returns the **bool** value **False** otherwise. Note that it cannot "refuse" or fail for any single argument that is passed to it; it simply returns True or False.

In a separate module **hw06_test.py**:

- import the **hw06.py** module,
- IN HONOR of having discussed == for the SECOND time, we are going to use it to now STREAMLINE at least some of our Python tests.

- The idea: we are going to directly compare each call to the desired value when we can, and only print the **result** of the comparison --- then, when you run a test module, if you see a bunch of **True**'s printed to the screen, then you know the tests pasts. Any **False's**, and you know that one or more tests failed.

- SO: write a **print** statement that says, **testing single_letter_key_dict**.
- THEN, write a **print** statement that says, **"True == Passed, False == Failed"**
- and then, write a print statement that compares a call to **single_letter_key_dict** to its expected value. For example,

```
print "Testing single_letter_key_dict:"
print "   (True == Passed, False == Failed)"
print ""
print hw06.single_letter_key_dict('George') == False
print hw06.single_letter_key_dict({'a':3, 'b':(1, 2, 3)}) == True
```

...for testing calls involving at least **2** arguments of **different** non-dict types, and at least **2** arguments that are dict's, at least one which returns True and at least one that return False.

2. Now, for a second predicate. Write a predicate function **all_int_values_dict** that takes any single argument, and returns the **bool** value True if that argument is a dict whose values are **all** integer values, and returns the

bool value **False** otherwise. Again, it cannot "refuse" any single argument that is passed to it; it simply returns True or False.

Add to module **hw06_test.py:**

- * write a **print** statement that says, **testing all_int_values_dict**.
- * THEN, write a **print** statement that says, **"(True == Passed, False == Failed)"**
- * and then, write a print statement that compares a call to **all_int_values_dict** to its expected value. For example,

```
print "Testing all_int_values_dict:"
print "   (True == Passed, False == Failed)"
print "-------------------------------------"
print hw06.all_int_values_dict('George') == False
print hw06.all_int_values_dict({'a':3, 'b':(1, 2, 3)}) == False
print hw06.all_int_values_dict({'a':3, 'b':148}) == True
```

...for testing calls involving at least **2** arguments of **different** non-dict types, and at least **2** arguments that are dict's, at least one which returns True and at least one that return False

3. Now, for the reason predicates in questions #1 and #2 exist...

Write a predicate function **is_letter_freq_dict** that returns the **bool** value True if its parameter is a dictionary consisting of keys which are single-letter strings and values which are integers; otherwise, it returns the **bool** value False. (That is, it tells us whether or not its argument is something structured like out letter-frequencies from HW #5.) As with #1's and #2's predicates,, it cannot "refuse" any single argument that is passed to it; it simply returns True or False.

Note that **is_letter_freq_dict** is **required** to appropriately use single_letter_key_dict and all_int_values_dict.

Add to module **hw06_test.py:**

- * write a **print** statement that says, **testing is_letter_freq_dict**.
- * THEN, write a **print** statement that says, **"(True == Passed, False == Failed)"**
- * and then, write a print statement that compares a call to **is_letter_freq_dict** to its expected value. For example,

```
print "Testing is_letter_freq_dict:"
print "   (True == Passed, False == Failed)"
print "-------------------------------------"
print hw06.is_letter_freq_dict('George') == False
print hw06.is_letter_freq_dict({'a':3, 'b':(1, 2, 3)}) == False
print hw06.is_letter_freq_dict({'a':3, 'bob':48}) == False
print hw06.is_letter_freq_dict({'a':3, 'b':148}) == True
```

...for testing calls involving at least **2** arguments of **different** non-dict types, and at least **3** arguments that are dict's, at least one which returns True and at least two that return False (for DIFFERENT reasons...)

I won't make you do anything with it here --- but can you see how this might type of predicate might improve the **robustness** of a function such as **freq_bar_chart**, that expects such a dict as its argument? It could call this predicate as its first action, and take appropriate (customized) action if its argument is inappropriate.

4. Since we don't have a lab exercise to do it --- we need something to dabble in references and copies and deep-copies, oh my. But we need a couple of helper functions to get us there.

FIRST: Write a predicate function **is_compound**. It returns **bool** value **True** if its argument is a list, tuple, or dict; it returns False otherwise. (We are not considering strings to be "compound", for our purposes here.)

Test it in **hw06_test.py** as you tested the functions from problems #1 - #3; test it on at least a list, a tuple, a dict, and on at least two arguments of different types that are neither list, tuple, nor dict.

5. Now write a predicate function **has_compound_components** that uses **is_compound** in its task of returning the **bool** value **True** if its single argument is a list, tuple, or dict that has at least one nested list, tuple, or dict within it. It returns the **bool** value **False** otherwise.

(Careful --- you need to check BOTH the keys AND the values, in the case of a **dict** argument...)

Test it in **hw06_test.py** as you tested the functions from problems #1 - #3; test it on at least:

* one list with a nested list-or-tuple-or-dict within it,
  and one list without;
* one tuple with a nested list-or-tuple-or-dict within it,
  and one tuple withou;
* one dict with a nested tuple as a key,
  one dict with a nested list-or-dict-or-tuple as a value,
  and one dict without either
* at least two arguments of different types that are neither list, tuple, nor dict.

6. Which finally brings us to **custom_copy**, which returns a copy of its argument meeting the following rather bizarre specifications (designed for feature-practice!):

* IF the argument is not a list, dict, or tuple, it simply returns the argument. No copying should be necessary.
* IF the argument is a list, dict, or tuple:
    * IF that list, dict, or tuple itself contains any lists, dicts, or tuples, it should return a **deep copy** (using the **copy** module's **deepcopy** function, as described in lecture and in Chapter 7 of "Learning Python")

      (hint: it works to call import within a function... if it gets called more than once in a Python session, we know that there's no effect from subsequent calls, anyway...)

    * IF it is a tuple or list that does NOT contain any lists, dicts, or tuples, it should return a **copy** using an empty-limit slice (as described in lecture and in Chapter 7 of "Learning Python").

    * IF it is a dict that does NOT contain any lists, dicts, or tuples, it should return a **copy** using the dict copy method.

custom_copy should appropriately use **is_compound** and **has_compound_components**, naturally.

This one is tricky to test; we'll kluge some "light"/wimpy tests in a way that gives us an excuse (I think) to use **is**. Test it in **hw06_test.py** as follows, on at least:

* one list with a nested list-or-tuple-or-dict within it; use **is** to show the returned value is NOT the same memory, and use == to show the returned value IS equivalent. That is, in **hw06_test.py**:

```
L1 = [1, [1, 2]]
```

```
L2 = hw06.custom_copy(L1)
print (L1 is L2) == False
print (L1 == L2) == True
```

and one list without any nested list-or-dict-or-tuple within; again, use **is** to show the returned value is NOT the same memory.

* one tuple with a nested list-or-tuple-or-dict within it,
  and one tuple without; use **is** and == as shown above.
* one dict with a nested tuple as a key,
  one dict with a nested list-or-dict-or-tuple as a value,
  and one dict without either; use **is** and == as shown above.
* at least two arguments of different types that are neither list, tuple, nor dict. Use **is** and == here, too, but be careful --- when should **is** be True for these, if ever? (This may depend on your arguments! 8-))

And, when you are satisfied, you should create a submittable output file **hw06_test.out**:

```
python hw06_test.py > hw06_test.out
```

By the due date and time given at the beginning of this handout, use **~st10/480submit** to submit your final versions of **hw06.py**, **hw06_test.py**, and **hw06_test.out** (And remember --- you can submit more than one version of these before the deadline, if inspiration strikes after a submission. As the syllabus notes, I'll simply grade the latest version that was submitted before the deadline.)