CIS 480 - Python - Fall 2005
WEEK 14 LAB EXERCISE and Homework #10

week 14 lab exercise due: Thursday, December 1st, END of lab
HW #10 due: Thursday, December 8th, 12:00 noon

FOR THE MODULE for HW #10:

...start each module with a **docstring** (as described in lecture) that includes the file name of the module, your name, the date the module was last modified, and any other documentation/description you'd like to provide.

For each function in a module, include at least a one-line **docstring** giving a brief description of that module.

Make sure that if you run the function **help** for your module or for a function within your module that you indeed see your docstrings incorporated.

WEEK 14 LAB EXERCISE

1.

(a) What exception is thrown if you try to import a module that doesn't exist or isn't accessible or otherwise isn't available?

____

(b) Assume that you are in a position where two modules contain a particular function **do_this** --- however, the module **first_choice.py** has a "better" version than the module **second_choice.py** contains.

Write a fragment of Python code <u>appropriately using a try-statement</u> that will try to make just **do_this** from **first_choice.py** available for use; however, if the exception <u>from part(a)</u> is thrown, then it will try to make just **do_this** from **second_choice.py** available instead.

(c) Let's take this one step further. Assume **first_choice.py** still has the "best" version of **do_this**, and **second_choice.py** still has the second-best version of **do_this**. HOWEVER --- if neither is available, you wish to define a quick-n-sleazy version of **do_this**, and at least have that available.

Write a fragment of Python code appropriately using NESTED try-statements that will try to make just **do_this** from **first_choice.py** available for use; however, if the exception from part (a) is thrown, then it will try to make just **do_this** from **second_choice.py** available instead; however, if the exception from part (a) is thrown THEN, it should define its own version of **do_this** instead. (For our practice purposes here, you can make **do_this** be a function that does, well, whatever you'd like.)

(d) What exception is thrown if you try to **index** a dict using a non-existent key?

____

2. You will recall that if you use an **except:** with a **try**, then it catches any exception (not yet caught by a previous, more "specific" **except**).

But --- what if you've used it, and you'd like to know WHAT exception you caught?

If you import module **sys**, then **sys.exc_type** will contain the TYPE of the exception throwm and **sys.exc_value** will contain the VALUE for the exception thrown.

for example:

```
>>> import sys
>>> def play(val):
...     try:
...         return val + 'hi'
...     except:
...         print "Type of Exception thrown:", sys.exc_type
...


>>> def play2():
...     try:
...         raise IOError, 13
...     except:
...         print "Type of Exception thrown:", sys.exc_type
...         print "Value of Exception thrown:", sys.exc_value
...
>>>
```

Type the above in. Then, try the expressions below, and write out what python results in for each:

```
>>> play('ho')
```

____

```
>>> play(13)
```

____

```
>>> play2()
```

____

____

HOMEWORK #10

1. Create a new module **hw10.py**.

Here is a small warm-up: Pretend, for a few moments, that you have forgotten that the dict **get** mathod exists.

While a dict method is clearly the more elegant option, you should find that one can write an analogous function that behaves similarly using a try-statement.

In module **hw10.py**, write a function **get_from_dict**. It should take two or three parameters: a dictionary (required), a desired key (required), and an "optional" default value to be returned if the given key is not a key in the given dictionary. (The third parameter's should be **None** --- not a string, but a NoneType literal --- if no third argument is given.)

get_from_dict MUST use a try-statement appropriately, and it MUST essentially behave like the dict **get** method, except that it is a function instead of a dict method.

That is, `get_from_dict(mydict, desired_key, default_val)`
should behave exactly the same as `mydict.get(desired_key, default_val)`

Likewise, `get_from_dict(mydict, desired_key)`
should behave exactly the same as `mydict.get(desired_key)`

HINT: you can do this in FIVE LINES, NOT counting the docstring!

2. Assume that, SOMETIMES, there is a file in your local directory **latest_choice.txt** that contains a string of interest.

In module **hw10.py**, write a function **grab_latest** in module **hw10.py** that tries to open a file named **latest_choice.txt** in the current directory and then read <u>and return</u> this string of interest; however, if this is not possible, it should NOT fail --- instead, it should interactively ask the user for the string of interest, and then write that resulting string of interest into a file named **latest_choice.txt** in the current directory, and <u>then return</u> this string of interest.

3. In support of problem #4, we need a little function. Fortunately, one of the ways that it can be written can involve using a try-statement.

In module **hw10.py**, write a function **get_arglist**. It takes no parameters; instead, it interactively requests potential arguments from the user. That is, it repeatedly asks the user to enter another argument, stopping when an argument that it an empty string is entered (which is what raw_input returns if you type the return/enter key immediately after the prompt).

Remember that raw_input always returns a string; so, you are <u>required to use try-statements</u> to convert the latest argument to an int or to a float, if possible; if neither is possible, however, it should be left as a string. Then, this resulting argument should be appended to an argument list, and when the user has finished entering arguments, these arguments should be returned as **get_arglist**'s return value.

(Yes, I know this can be done in other ways --- for example, using regular expressions. But this is a homework whose purpose is to practice exception-handling, so use of try-statements is required.

For example:

```
>>> hw10.get_arglist()
Enter an argument: 3
Enter another argument, or hit return to stop: 27.3
Enter another argument, or hit return to stop: hi
Enter another argument, or hit return to stop: 7
Enter another argument, or hit return to stop:
[3, 27.300000000000001, 'hi', 7]
```

4. Now we'll play around with a little exception-tester, that incorporates an idea or two from earlier in the semester as well as some exception-handling ideas.

In **hw10.py**, write a function **exc_tester**. It expects one parameter, a **function**. It should use **get_arglist** from problem #3 to obtain a list of arguments from the user, and then use the **apply** function to apply the parameter function to the argument list returned by **get_arglist** --- HOWEVER, it should do this in such a way that, IF an exception is generated in applying this function to this argument list, **exc_tester** should catch it and, instead of failing, print a pleasant message giving the exceptions type and, if it exists, its value.

5. Sigh --- we really haven't done anything with the **raise** statement yet! So: for the **final 10 points** of this assignment, write a function **raise_demo** in **hw10.py** that does SOMETHING interesting with at least one **raise** statement. It cannot be straight from the lecture, from the lab exercise, or from the text; it can be an interesting variation from these sources, though.

Perhaps it could use raise in printing a soothing or informative message before passing on an exception to a higher level; perhaps you could set up a custom-exception of your own, and the function could raise it when appropriate.

Any attempt will garner you at least 5 points of the possible 10 for this problem.

6. 5 POINT BONUS:

Something is just not right with the following function ---
(see next page)

```
def price_check(unit_price, quantity):
    assert (quantity >= 0, "cannot have a negative quantity")
    assert (unit_price >= 0, "cannot have a negative unit price")

    print "You would owe $%.2f for %d of this item." %\
          (unit_price * quantity, quantity)
```

Sure, when you call `price_check(3, 4)`, it prints:

```
You would owe $12.00 for 4 of this item.
```

But, when you call `price_check(-3, 4)`, it doesn't fail! It prints:

```
You would owe $-12.00 for 4 of this item.
```

For a 5 point bonus --- **explain WHY** this behavior is occurring, and how it can be fixed. Type your answer in a triply-quoted string within **hw10.py**, followed by a corrected version of **price_check**.

Sadly, there is no time to come up with proper testing specifications for a **hw10_test.py** module, so none is required for this homework. You should, of course, still test all of your functions thoroughly!

By the due date and time given at the beginning of this handout, use **~st10/480submit** to submit your final version of **hw10.py** (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.)