CIS 130 - Intro to Programming - Spring 2007
Homework Assignment #4 - **INDIVIDUAL** assignment

Homework #4 DUE**: BEGINNING** of class, Wednesday, February 28, 2007

Purpose: get practice with functions with side-effects, interactive input, and interactive output. There's some more if-statement practice in here, too.

How to turn in: use the tool **~st10/130submit** on cs-server to turn in the files **hw4.py** and **hw4.txt** that you create below.

CONSIDER:

* You can use + to produce a new string that is the concatenation, or merged result, of two strings; that is, + allows you to concatenate two strings when it is called with two string operands:

```
>>> name1 = "Sarah"
>>> name2 = "Stout"
>>> name1 + name2
'SarahStout'
```

  * HOWEVER - + can only be used to concatenate two STRINGS. That is, trying to concatenate a string and a number (or trying to add a number to a string? 8-)) fails:

```
>>> name1 + 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

  * Happily, it is easy to convert a number to an equivalent string: **str(**expr) tries to return a string equivalent for the **expr** passed to it.

```
>>>  type(3)
<type 'int'>
>>> type( str(3) )
<type 'str'>
>>> str(3)
'3'
>>> str(True)
'True'
```

  * SO:

```
>>> name1 + str(3)
'Sarah3'
```

* In a related comment, remember that raw_input always returns what the user types as a string.

  What if you'd like to treat that string as, say, a number? (Perhaps the user typed 13, which raw_input returns as '13'?)

int(expr) will try to return the integer corresponding to **expr** (if it can);
float(expr) will try to return the float value corresponding to **expr** (if it can).

* These are all useful to know in functions that use interactive input and output...

NOW, Consider the following function:

```
#-----
# Contract: get_hypotenuse: number number -> number
# Purpose: compute the length of the hypotenuse of a right
#          triangle whose other two sides are of length <side1>
#          and <side2>
# Examples: get_hypotenuse(3, 4) should be 5.0
#           get_hypotenuse(3, 7) should be 7.62
#-----

from math import *

def get_hypotenuse(side1, side2):
    return sqrt( pow(side1, 2) + pow(side2, 2) )
```

Now that we have functions with side effects, you should be able to see how we might write a function that simply runs all of get_hypotenuse's examples:

```
#-----
# Contract: test_get_hypotenuse: void -> void
# Purpose: run the examples for function get_hypotenuse, comparing
#          each call to the expected result of that call, and printing
#          the result of that comparison.
# Examples: test_get_hypotenuse() should cause the following to be
#           printed to the screen:
#
# Testing get_hypotenuse (True means a passed test):
# -------------------------------------------------
# True
# True
#-----

def test_get_hypotenuse():
    print ""
    print "Testing get_hypotenuse (True means a passed test):"
    print "-------------------------------------------------"
    print get_hypotenuse(3, 4) == 5.0
    print abs(get_hypotenuse(3, 7) - 7.62) < .001
```

This may not be a very dynamic function, but if you write it early on, you can use it to quickly test get_hypotenuse while it is being tested and debugged; and, if you decide later to change or modify get_hypotenuse, it is very easy to re-run all the examples (as you should do ANY time you modify a function!)

A more interesting function with side-effects might be an interactive "shell" for get_hypotenuse; it could interactively ask the user for the appropriate values, then call get_hypotenuse with those values, and

print out the result. Or,

```
#-----
# Contract: ask_get_hypotenuse: void -> void
# Purpose: provide an interactive "shell" for get_hypotenuse;
#          ask the user for the expected right triangle non-hypotenuse
#          side lengths, then print to the screen the corresponding
#          hypotenuse length
#
# Example: if someone calls get_hypotenuse and enters 3 and 4 when
#          prompted, the following should be printed to the screen:
#
# The hypotenuse's length is: 5.0
#-----

def ask_get_hypotenuse():
    side_a = raw_input(
        "Enter the length of one non-hypotenuse side of a right triangle: ")

    side_b = raw_input(
        "...and now the length of the other non-hypotenuse side: ")

    # since raw_input returns a string, need to convert these hopefully-numeric
    #     values to numbers to be able to call get_hypotenuse with them

    side_a_num = float(side_a)
    side_b_num = float(side_b)

    hyp_length = get_hypotenuse(side_a_num, side_b_num)

    print "The hypotenuse's length is: " + str(hyp_length)
```

So, getting on to the problems:

Type all of the functions for this assignment in a file **hw4.py**.

1. NOTE that it is considered POOR STYLE for a "pure" function such as those we have been writing before this week to do interactive i/o; these functions are expected to quietly take their arguments, compute a result, and return it to the caller. They can be easily called from other programs without messing up those programs' user interfaces, causing unexpected output to the screen, etc.

But, some functions are written **for** such side-effects; their whole point is to have an effect, not necessarily return a value.

Write a function **show_sig** that takes no parameters, and returns no result; it simply prints to the screen **at least three lines** that include your name and whatever else you'd like to include. (For example, how might you "sign" an e-mail message? You might include your name, that you are a student at Humboldt State University, and perhaps a message of the day.)

I'll leave it up to your what your "sig" should include, as long as it meets the following minimum requirements:

* it should include your name;
* you should print something non-blank on at least three different lines;
* you should try to make it attractive.

In **hw4.txt**, paste in part of a **python** session in which you run **show_sig**, showing the call and what it prints to the screen.

2. Consider the function **semester_grade** from HW #2, question #4. (Note that a version of it is also available from the course Moodle site.)

Copy **semester_grade**'s function definition into **hw4.py**, and add examples until you have at least three examples. Then design and write a canned-examples test function **test_semester_grade** in the same style as **test_get_hypotenuse**, that runs each of those example calls, compares its result to the expected result, and **prints** True or False based on if the actual result is close enough to the expected result.

Be sure to include a "Testing..." string printed to the screen, saying what function is being tested and explaining that each True seen means a passed test.

In **hw4.txt**, paste in part of a **python** session in which you run **test_semester_grade**, showing the call and what it prints to the screen.

3. Let's say you would now like a nice interactive shell for **semester_grade**. Design and write **ask_semester_grade**, in the same style as **ask_get_hypotenuse**, that will expect no arguments and will return no result, but that will ask the user to input the three desired averages (be specific in your prompts! The different averages have different weights in the semester grade, remember.)

Then it should do what it has to to call get_hypotenuse with the values the user has typed in, and then print to the screen a descriptive message saying what the hypotenuse of this triangle is.

In **hw4.txt**, paste in part of a **python** session in which you run **ask_semester_grade**, showing the call and the whole dialogue - what it prints to the screen, what you then enter, and what it finally prints to the screen.

4. Note that **lect03.py** (on the in-class examples portion of the public course web page) includes functions for square_area, rect_area, circle_area, and ring_area.

Create a copy of **lect03.py** in your current working directory, and then you will be able to **import** its functions into **hw4.py**. Do so, and then:

Create a function **which_area** which asks the user to type **1** if they want a square area, **2** if they want a rectangle area, **3** if they want a circle area, and **4** if they want a ring area.

If the user types **1**, the program should ask for the side length, and then call **square_area** to compute such a square's area, and then print the result to the screen in a descriptive message. If the user types **2,** the program asks for the two rectangle sides, and then calls **rect_area** to compute such

a rectangle's area, and then prints the result to the screen in a descriptive message. And so on, for **3** and **4** as well.

What if the user types something BESIDES **1** through **4**? The program should simply complain to the screen, in that case, saying that an illegal answer was given.

Examples section tip: I'll be satisfied if, for each example, you describe specifically some explicit values the user would type in, and then show what **which_area** should print to the screen for that case. (That is, you don't have to reproduce the actual prompt text in each example, just a description of what values the user inputs in that scenario and the final result each example scenario would print to the screen - like you see in **ask_get_hypotenuse** at the beginning of this handout. PLEASE ASK me if this is not clear!)

You do not have to submit in **lect03.py**, note.

In **hw4.txt**, paste in part of a **python** session in which you run **which_area** for each of its examples (how many at the very least?), showing for each the calls and the whole dialogue - what it prints to the screen, what you then enter, and what it finally prints to the screen.

5. (This is **if**-statement practice, and hopefully practice making use of the new modified design recipe.)

(Adapted from www.htdp.org) A manufacturing company measured the productivity of its workers and found that between the hours of 6 am and 10 am --- that is, from 6-7, 7-8, 8-9, and 9-10 --- they could produce 30 pieces/hour/worker;

between 10 am and 2 pm --- from 10-11, 11-12, 12-1, and 1-2 --- they could produce 40 pieces/hour/worker;

and between 2 pm and 6 pm --- from 2-3, 3-4, 4-5, and 5-6 --- they could produce 35 pieces/hour/worker.

At all other hours, 0 pieces are produced per hour per worker --- the company is closed then.

Write a function **pieces_produced** that takes as its arguments an hour of the day expressed in twenty-four hour format, along with the number of workers working during that hour; it computes and returns the total number of pieces produced by that many workers during that hour of the day. (A first argument of 15, then, is asking how much the given number of workers produces working from 15:00-16:00, or from 3-4 pm.)

You will be writing testing functions for this function next...

6. Now design and write a canned-examples test function **test_pieces_produced** in the same style as **test_get_hypotenuse**, that runs each of **pieces_produced**'s example calls, compares its result to the expected result, and **prints** True or False based on if the actual result is close enough to the expected result.

Be sure to include a "Testing..." string printed to the screen, saying what function is being tested and explaining that each True seen means a passed test.

In **hw4.txt**, paste in part of a **python** session in which you run **test_pieces_produced**, showing the call and what it prints to the screen.

7. And it is easy to imagine someone wanting to play around with this function, perhaps. So, to make such play a little easier, let's create an interactive shell for **pieces_produced**. Design and write **ask_pieces_produced**, in the same style as **ask_get_hypotenuse**, that will expect no arguments and will return no result, but that will ask the user to input the values required, and then call **pieces_produced** appropriately, then printing to the screen a descriptive message indicating how many pieces would be produced.

In **hw4.txt**, paste in part of a **python** session in which you run **ask_pieces_produced**, showing the call and the whole dialogue - what it prints to the screen, what you then enter, and what it finally prints to the screen.

When you are happy with your files **hw4.py** and **hw4.txt**, type the following command at the cs-server prompt:

~st10/130submit

Then follow its directions to submit your files .