CIS 130 - Intro to Programming

NOT REALLY A Week 8 Lab Exercise

Purpose: get more practice with sentinel-controlled loops

Answer the following in the space provided:

1. Consider **asking_profits.py** (available along with this handout). It uses a sentinel-controlled loop to ask a user for a ticket price, and then prints to the screen the profit for that ticket price; it continues until the user enters a ticket price of **-1** (that's the **sentinel** value, in this case).

Read this over carefully, and make sure that you understand what it is doing and how it is doing it.

Then, copy it into a file in your current working directory with the name **asking_profits.py**. If you need to, also copy over **profit_ex.py** (which contains the profit/revenue/cost/attendance combo).

(a) Run **asking_profits** in the **python** interpreter, and immediately enter a ticket price of **-1**. What gets printed to the screen as a result?

(b) Run **asking_profits** again, and type at least three "real" ticket prices before typing in a ticket price of **-1** to stop. Below, write the ticket price you tried, and the profit it printed. out for it:

ticket price: _____ profit: _____

ticket price: _____ profit: _____

ticket price: _____ profit: _____

Now enter a ticket price of **-1** to end the program. Does the program try to print the profit from a ticket price of **-1**?

_____

2. A little food for thought: in programming, there is almost always more than one way to solve a problem. Some of these may be equally reasonable - others may work, but in a less-elegant or otherwise less-desirable way.

Consider the following two functions. As you can see, **sum_all1** uses a classically-structured sentinel-controlled loop:

```
def sum_all1():

    SENTINEL = -1
    input = 0
```

```
    sum = 0

    input = int(raw_input("enter the first input, or " + str(SENTINEL)
                          " to stop: "))

    while (input != SENTINEL):

        sum = sum + input

        input = int(raw_input("enter the next input, or " + str(SENTINEL)
                              " to stop: "))

    print "sum is: " + str(sum)
```

now consider **sum_all2**, which uses an alternative approach:

```
def sum_all2():
    SENTINEL = -1
    input = 0
    sum = 0

    while (input != SENTINEL):

        input = int(raw_input("enter the next input, or " + str(SENTINEL)
                              " to stop: "))

        if (input != SENTINEL):
            sum = sum + input

    print "sum is: " + str(sum)
```

Say that someone wants to use these to add up 50 inputs.

For **sum_all1**, how many values are typed in by the user? _____

For **sum_all2**, how many values are typed in by the user? _____

For **sum_all1**, how many times will you compare **input** to SENTINEL while handling these input values?

_____

For **sum_all2**, how many times will you compare **input** to SENTINEL while handling these input values?

_____

Both work - but **sum_all2** requires about two times the number of comparisons that **sum_all1** does. This just feels more redundant, as well as more "clunky", than it needs to be, when the classic sentinel structure makes half of those comparisons unnecessary.

3. A little Boolean-play, in preparation for some of HW #6:

Consider the following truth table:

A	B	(A and B)	(A or B)
False	False	False	False
False	True	False	True
True	False	False	True
True	True	True	True

What is the opposite of (A and B)? You know that it is not(A and B):

A	B	(A and B)	not(A and B)
False	False	False	True
False	True	False	True
True	False	False	True
True	True	True	False

If two expressions have the same set of values in their truth table columns, then they are equivalent expressions.

So: complete the following truth table (and feel free to use the python interpreter to check your entries). See which Boolean expression is equivalent to not(A and B):

A	B	not A	not B	(not A) and (not B)	(not A) or (not B)
False	False	_____	_____	_____	_____
False	True	_____	_____	_____	_____
True	False	_____	_____	_____	_____
True	True	_____	_____	_____	_____

Which expression above is equivalent to not(A and B)? _____

Likewise, consider not(A or B):

A	B	(A or B)	not(A or B)
False	False	False	True
False	True	True	False
True	False	True	False

True True True False

Look back up at the truth table you completed above. Which expression turns out to be equivalent to not(A or B)?

_____

Above, you have actually proven deMorgan's Laws. And if you keep this in mind, it can be useful in writing the precise conditions you want for if-statements and while-loops.

How so? Well, someone asked in class the other day if you could write code that would give users another chance if they entered a value known to be unreasonable - for example, if you ask them to enter y or n (for yes or no), and they enter something else.

But - what if the user answered something else again? a third time? a fourth? You need repetition, to ensure that you keep trying until something "legal" is entered. You could do that with a kind of sentinel-controlled loop, true? You could keep looping while the user has entered something that ISN'T y or n...

Write a Boolean condition would be true if a variable value_entered is 'y' or 'n':

_____

Considering what you've learned above, you should now be able to determine one of the several expressions that will then correctly be true if a variable value_entered is NOT 'y' or 'n':

_____

3. So, now write **answer_y_or_n**, a function which expects a string as a parameter, the (presumably yes-no) question to be asked, which keeps asking that question and reading in what the user types as a result until he/she types a legal 'y' or 'n'; it then returns that 'legal' response as its value.

Since this is not actually a lab exercise, I've posted a copy of this with answers included on the course Moodle site, under "Some solutions." I would recommend filling this out yourself, and then comparing your answers with those posted.

And - if you have any questions about those answers - then please be sure to ASK ME about them.