CS 1110, LAB 6: ASSERTS AND DEBUGGING

http://www.cs.cornell.edu/courses/cs1110/2017fa/labs/lab6/

First Name: _____ **Last Name**: _____ **NetID**: _____

Now that you have finished a long project like Assignment 1, you see just how common errors can be. While you have a lot of experience with testing now, the tests do not always tell you where the error is or how to fix it.

This lab is meant to give you practice with debugging. We have created some buggy functions, and you need to find the errors. In the second part of the lab, you will work with asserts. You will see how to use them in defensive programming.

Warning: This lab will have a lot of short-answer questions and the online system will be very picky about what constitutes a correct answer. If you are having trouble getting the online system to accept your answers, use the PDF instead.

1. Lab Files

For today's lab you will need the following additional file:

- `lab06.py` (a collection of buggy functions)

As always you should create a new directory on your hard drive and download this file into that directory. This lab will require that you modify this file, as well as answer some questions about it. Even if you are using the online system, work directly with this file. **Do not try to type your answers into the system before you have verified that they are correct.**

Getting Credit for the Lab. Once again, you have a choice between getting credit through the online system or your instructor. The online lab is available at the web page

http://www.cs.cornell.edu/courses/cs1110/2017fa/labs/lab6/

If you use this worksheet, your answer will include both a sheet of paper (or the sheet provided to you in lab) and the file `lab06.py`. When you are finished you should show both to your lab instructor, who will record that you did it.

As with all previous labs, if you do not finish during the lab, you have until the **beginning of lab next week to finish it**. Over the next week, you may either (1) complete the lab online, (2) show your lab to a consultant during consulting hours, or (3) show your lab to an instructor at the *beginning* of the next lab session.

Course authors: D. Gries, L. Lee, S. Marschner, W. White

2. Debugging (Part 1)

Last week you played with the `Time` object. This week we have two time-related functions. However, instead of storing time as an object, these functions process time as a string. This is much more error-prone that working with `Time` objects. Strings do not verify that the hours and minutes are in range. That is why these two functions have several errors in them.

The function `time_to_military` assumes that the argument is a string in 12 hour format. In this representation, the hour is a number between 1 and 12, and the string ends in 'AM' or 'PM' (capitalized). For example, '9:05 AM' is in 12 hour format, as is 10:45 PM'. However, international/military time like 13:55' is not, nor are typos such as '9:05PM'.

This function has a bug inside of it. However, instead of staring very hard at the code, we want you to approach debugging in a much more organized fashion.

This function converts time in 12 hour format to international/military format. Read the specification in `lab06.py` for an understanding of how it works. Note that international/military time has a leading '0' if the hours are less than 10.

You cannot do debugging without testing. So the very first thing you need to do is to come up with some test cases. Think of how the function is going to break up the input into hours, minutes, and times of day. Also pay close attention to input values that are "on the border" (e.g. midnight). With that in mind, list at least six interesting tests that you could try. **We do not want you to write a test script or try to fix any bugs**. Just write down the test cases.

Input	**Expected Output**

Now you want test these functions and look at the program flow. For this function, we are going to use the Python Tutor. The Python Tutor shows you which lines are skipped and which lines are executed. It also shows you all of the values of the variables.

Copy the function from `lab06.py` into a tab in the Python Tutor. At the bottom, add function calls for your test cases above. You do not need cornelltest. You can just print the output of the function call and visually see if it matches the expected result.

Run through each of your test cases, stepping through the call one line at a time. There are three errors in this program. If you do not see them, pick some more test cases and try again. When you find all the errors, write them below. In your answer, include **both the line number of the**

error and the entire line of code for that error. The line numbers should be the ones the file `lab06.py`; **do not use the line numbers from the Python Tutor**.

Now you can fix these errors in `lab06.py`. When you fix them, you should only change lines; you should not need to add any new lines. Adding new lines can changes your answers to the next part, particularly in the online system.

3. Debugging (Part 2)

The function `time_to_minutes` is similar to `time_to_military` in that it assumes that the argument is a string in 12 hour (not military) format. The main difference between this function and `time_to_military` is that it returns the number of minutes (as an `int`) since midnight. Like `time_to_military`, this function also has a bug in it. But this time, instead of using the Python Tutor, you will use watches and traces to find the bugs.

Because `time_to_minutes` takes the same input as `time_to_military`, you can reuse the test cases that you used for that function. The outputs will be different, but you can use the same inputs that you used before.

However, this time we do not want you to use the Python Tutor. We want you to test the program normally using the Python interactive shell. Import `lab06.py` and test this program by calling it, as shown below.

```
>>> import lab06
>>> lab06.time_to_minutes('11:05 AM')
```

What do you see when you do this?

As you can see, Python is trying to help you find the error by giving you some line numbers. However, Python is not particularly smart. Sometimes it finds the error too late. So even if it told you the error occurs at a certain line, the real error is often at an earlier line.

To help us understand what is going on, we need to add traces and watches (covered in class). Add them where you think they go. As a general rule, we like to add a watch after every assignment statement, and a trace after every line that might be skipped. Once you have done that, repeat the function call above. Hopefully you will see the error now.

What is the error? Once again, **indicate the original line number and the entire line of code for the error.**

Keep the traces and watches there. Now try all the inputs that you used for `time_to_military`. You should be able to identify two more errors. What are they? Once again, **indicate the original line number and the entire line of code for the error.**

Once again you can fix these errors in `lab06.py`. Do that before continuing.

4. Asserts and Preconditions

Strings have several methods that allow us to check their contents. Here are a few of them:

Method	Description
s.isalpha()	**Returns**: True if `s` is nonempty and has only letters; False otherwise.
s.isdigit()	**Returns**: True if `s` is nonempty and has only numbers; False otherwise.
s.isalnum()	**Returns**: True if `s` is nonempty and has only numbers or letters; False otherwise.
s.islower()	**Returns**: True if all letters in `s` are lower case; False otherwise.
s.isupper()	**Returns**: True if all letters in `s` are upper case; False otherwise.

These can be very helpful in asserting preconditions. As we saw in class, the contents of an assert statement must be a boolean expression.

Recall the function pigify from the previous lab. It had the following (abreviated) specification:

```
def pigify(w):
    """
    Returns: copy of w converted to Pig Latin.
    Precondition: w is a nonempty string with only lowercase letters
    """
```

You do not need your implementation (in `lab05.py`) for this lab. However, you should reread the precondition. In the box below, write one or more assert statements that enforce this precondition. Remember to check the type as well!

Next, you are to assert the preconditions for the functions `time_to_military` and `time_to_minutes`. However, that is much harder. As you can tell when you created your test cases, that simple-looking precondition is quite complex. Therefore, we have added the function `is_time_format` to `lab06.py`. This function takes that parameter `s` and returns True only if `s` is a string in 12 hour format.

Look at the precondition for this function. There is none! This function must be prepared for any inputs. Implement this function in `lab06.py`. You may use as if-statements as you want. When you are done, you should test that the function works, though we do not need you to create a test script or list any test cases.

Use the function `is_time_format` inside an assert statement to enforce the preconditions of the functions `time_to_military` and `time_to_minutes`. Now call `time_to_military` on an invalid input like '23:15 PM'. Copy what you see below.

5. Recovering from Errors

There is one last function inside `lab06.py`. The function `something_to_military` is exactly like `time_to_military` except that there are **no preconditions on the function**. Instead, this function returns an error message (as a string) when the precondition to `time_to_military` is violated.

Implement the function `something_to_military` in `lab06.py`. You are not allowed to use if-statements. Instead, you should use a try-except statement and `time_to_military` as a helper. This will only work if you have successfully completed the assert statements from the previous part of the lab.