

CS 1110, LAB 6: ASSERTS AND DEBUGGING
<http://www.cs.cornell.edu/courses/cs1110/2016fa/labs/lab06/>

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.

Lab Materials. This lab will require an addition file called `lab06.py`. You can download this file from the Labs section of the course web page.

<http://www.cs.cornell.edu/courses/cs1110/2016fa/labs>

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 on this worksheet. You need to show both to your instructor to get credit for the lab.

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/cs1110/2016fa/labs/lab06/>

The advantage of the online system is that you can do it on your own time and verify that you got credit. The disadvantage is that your answers must be correct. If you want more guided help on this lab, you should use the worksheet instead. Despite the demands of the online system, labs are graded on effort, not correctness.

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**. You should always do your best to finish during lab hours.

1. DEBUGGING

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 than 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 functions `time_to_military` and `time_to_minutes` assume 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'.

These functions each have bugs in them. However, instead of staring very hard at the code, we want you to approach debugging in a much more organized fashion.

The function `time_to_military`. 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.

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. Include the line numbers from the file `lab06.py`.

Fix these errors before going on to the next function.

The function `time_to_minutes`. This function takes the time in 12 hour format and returns the number of minutes since midnight. Because it 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 in this program.

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 it?

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?

2. ASSERTS AND PRECONDITIONS

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

Method	Description
<code>s.isalpha()</code>	Returns: True if <code>s</code> is nonempty and has only letters; False otherwise.
<code>s.isdigit()</code>	Returns: True if <code>s</code> is nonempty and has only numbers; False otherwise.
<code>s.isalnum()</code>	Returns: True if <code>s</code> is nonempty and has only numbers or letters; False otherwise.
<code>s.islower()</code>	Returns: True if all letters in <code>s</code> are lower case; False otherwise.
<code>s.isupper()</code>	Returns: True if all letters in <code>s</code> 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 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, we want you 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. You may use as if-statements as you want. Test that the function works, though we do not need you create a test script or list any test cases.

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

3. 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 when the precondition to `time_to_military` is violated.

Implement this function. **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.

When you have completed your implementation of this function, you are done.