**EXERCISE 4: ASSERTS AND DEBUGGING**

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 exercise 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.

1. **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' (capital-ized). 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 specifica-tion in lab04.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 |
| 10:56 AM | 10:56 |
| 0:45 PM | 12:45 |
| 6:50 AM | 06:50 |
| 9:53 PM | 21:53 |
| 3:59 AM | 03:59 |
| 12:00 AM | 00:00 |
| 12:00 PM | 12:00 |

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 lab04.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 lab04.py; do not use the line numbers from the Python Tutor.

* Incorrect handling of AM time: **line 49***‘hour = 0’*
* Incorrect conversion of PM time: **line 47***‘hour += 12’*
* Incorrect leading zero handling: **line 52***‘if (hour <= 10)’*

Now you can fix these errors in lab04.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.

1. **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 lab04.py and test this program by calling it, as shown below.

* import lab04
* lab04.time\_to\_minutes('11:05 AM')

What do you see when you do this?

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "E:\OneDrive - ptit.edu.vn\pro\python\labs\lab04\lab04.py", line 96, in time\_to\_minutes

mins = int(mins)

ValueError: invalid literal for int() with base 10: ':05'

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.

* Variable name mistyped (*hoar* instead of *hour*): **line 86**

*‘hoar = hour+12’*

* Incorrect slicing to get *mins* value: **line 91***‘mins = s[pos1:pos2]’*

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.

* Incorrect conversion of PM time: **line 86***‘hoar = hour+12’*
* Incorrect minutes conversion: **line 91***‘mins = s[pos1:pos2]’*

Once again you can fix these errors in lab04.py. Do that before continuing.

1. **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!

assert isinstance(w, str), "Input must be a string"

assert len(w) > 0, "Input must be a nonempty string"

assert w.islower(), "Input must contain only lowercase letters"

assert w.isalpha(), "Input must contain only letters (no numbers or special characters)"

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 lab04.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 lab04.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.

Traceback (most recent call last):

  File " e:\OneDrive - ptit.edu.vn\pro\python\labs\lab04\lab04.py", line 165, in <module>

    print(time\_to\_military('23:15 PM'))

  File " e:\OneDrive - ptit.edu.vn\pro\python\labs\lab04\lab04.py", line 35, in time\_to\_military

    assert is\_time\_format(s), 'Invalid time format'

AssertionError: Invalid time format

1. **Recovering from Errors**

There is one last function inside lab04.py. The function something\_to\_military is exactly like time\_to\_military except that there are no preconditions on the function. Instead, this func-tion returns an error message (as a string) when the precondition to time\_to\_military is violated.

Implement the function something\_ to\_military in lab04.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.