Lab 4

CS161

For each of the problems below be sure to design your solution before you write any Python code. Follow the software development stages:

1. Analyze the problem
2. Determine the specifications for your solution
3. Create a design for your solution in pseudocode
4. Implement your design in Python
5. Test/debug your solution

Each Python module you turn in must begin with a block, called a *docstring*, that looks like the example below (see PEP 257 for more details). The module docstring must follow the shebang and coding statements in your module; i.e. it must be the first actual line of code in the module.

In your module docstring show the work you did following the software development stages above. Make notes about the problem as you analyze it; write specifications for what your program must do to find a solution; write your program out in pseudocode *before* you start to write Python; and design a set of test data that you can use to prove your program is producing correct results. Only after you've done all of this should you try to translate your pseudocode to Python.

```
#! /usr/bin/env python3
# coding=utf-8

"""
Brief, prescriptive, one sentence description of what the module does.

A more detailed explanation of what the module does. Use complete
sentences and proper grammar. Treat this as if it is documentation someone
else will be reading (because it is). Include any notes from your problem
analysis as well as your program specifications here.

Pseudo code, to be written before any actual Python code.

Assignment or Lab #

Firstname Lastname
Firstname Lastname of partner (if applicable)
"""
<two blank lines>
<your code>
```

Every function or method you write must also include its own docstring. That header will also use triple quotes and begin with a prescriptive, one sentence description of what the function or method does. If the function or method takes arguments, they should be described as well as any non-obvious return value(s). See examples below or PEP 257 for more details).

```
def print_greeting():
    """Print a greeting to the user."""
    print("Hello out there in userland!")


def hypotenuse(side_a, side_b):
    """
    Calculates the length of the hypotenuse of a right triangle using
    the formula c^2 = a^2 + b^2.

    side_a - the length of side A of the triangle
    side_b - the length of side B of the triangle
    """
    c = a**2 + b**2
    return math.sqrt(c)
```

Pair Programming

In this lab we are going to use **pair programming**.

> *Pair programming is an agile software development technique in which two programmers work together at one workstation. One, the **driver**, writes code while the other, the **observer** or **navigator**, reviews each line of code as it is typed in. The two programmers switch roles frequently.*
>
> *While reviewing, the observer also considers the "strategic" direction of the work, coming up with ideas for improvements and likely future problems to address. This is intended to free the driver to focus all of their attention on the "tactical" aspects of completing the current task, using the observer as a safety net and guide.*

Choose one partner in the lab to work with. You can pair program in one of two ways:

1. Two people can work at one computer, occasionally switching the driver and observer roles. It is *critical* that both members be engaged in the work. You should take turns being in the driver and observer roles with only the driver typing and the observer making comments or driving the direction of the code.
2. Two people can work at separate machines using Visual Studio Live Share. Live Share, available as a free plug-in for VS Code and already installed on the school's Surface devices, enables you and your partner to collaborate on the same codebase without the need any difficult configuration. When you share a collaborative session, your partner sees the context of the workspace in their editor. This means they can read the code you shared without having to clone a repo or install any dependencies your code relies on. They can use rich language features to navigate within the code; not only just opening other files as text but using semantic analysis-based navigation like Go to Definition or Peek. Again, you should take turns being in the

driver and observer roles with only the driver typing and the observer making comments or driving the direction of the code.

Exercise 1

This lab exposes you to a recommended software engineering method called "Design by Contract," while also providing practice in defining and using functions and in using PythonTutor to discover how a function behaves. Finally, it also introduces some very basic file input.

The Problem

You will work on some functions that might be used in a text formatting application. You *must* write your own version of these functions, you *may not* simply call standard library functions to do the work for you. You will be given a function for padding a string of words with spaces in order to justify the words in a line of a given length, together with a main function to use in testing. This function relies on some "helper functions" which you will write to the specifications given in each function's docstring. (See Part 1 below.)

Part 1

Download the starter python (.py) files and the text (.txt) files from the class shared folder to a directory on your computer. Open the text files to see what they look like. You will use these files in testing your final program. Each line of these files begins with a non-negative integer n followed by n "words" (consecutive non-whitespace characters); in the case of multiple words, a single space separates consecutive words. The last line of the files contains just the integer 0. These files are examples of intermediate files that a text formatting application might produce from a document that is to be formatted.

Load lab_4.py into your editor. This file defines two functions: a main function for you to call without any arguments to run tests and a function pad_words for main to invoke. The other two functions are just "stubs"—they return trivial results of the form needed by main and pad_words, but not the results needed for pad_words and main to behave correctly.

Check that you are able to execute the code in lab_4.py. Begin by calling the main function by simply entering main() in the Python shell. When prompted for a file name, type the name of one of the text files (either will do). When prompted for a line length, enter any integer. (Because the stubs don't use the input parameters for anything, it doesn't matter how big the integer is.) In its current form the program simply prints some blank lines.

Now look at the contents of lab_4.py. The functions in this file are designed using the software engineering method called Design by Contract (DBC). DBC is used in development of complex software by teams of programmers. The idea is to design functions using contracts, which precisely state (1) what a function can assume about the arguments it is called with and (2) what effect executing the function has. The statement of (1) is called a pre-condition and is usually labeled using the word "Requires." The statement of (2) is called a post-condition. We won't need the full generality of post-conditions for this lab; so instead of a post-condition, we just describe the results the function returns.

In lab_4.py, a function's contract is described in a function docstring. The function is correct if it returns correct results when the arguments passed to it satisfy the precondition—if they don't, the caller has

violated the contract and cannot count on the function returning any particular result. However, as long as both the function and the caller satisfy the contract, the program will exhibit correct behavior. DBC produces simpler function definitions because the function does not have to perform error checking because they assume that the contract is adhered to. The contracts also aid understanding and reasoning about correctness.

You will see how DBC works by writing bodies for the helper functions that satisfy the contracts given by their docstrings, and then executing main.

First, write the body for get_and_strip_number. Do not check that the input string has the form described in the pre-condition (following "Requires") since your function needs to work correctly only when is it called with correct arguments. However, be sure it returns the values described in the function's contract (following "Returns") when it is called with correct arguments. String slicing and the method find may be useful.

To clarify the contract, some sample calls and the results they should produce are:

```
>>> get_and_strip_number("4 This is a test.")
(4, 'This is a test.')
>>> get_and_strip_number("1 yikes!")
(1, 'yikes!')
>>> get_and_strip_number("0")
(0, '')
>>>
```

Next, write the body for get_and_strip_word. It should return the values described in the function's docstring when it is called with correct arguments. Some sample calls and their results are:

```
>>> get_and_strip_word("This is a test.")
('This', 'is a test.')
>>> get_and_strip_word("is a test.")
('is', 'a test.')
>>> get_and_strip_word("a test.")
('a', 'test.')
>>> get_and_strip_word("test.")
('test.', '')
>>> get_and_strip_word("")
('', '')
>>>
```

Now execute main. At the prompts give "gettysburg.txt" and the number 59 or more, or give "test_file.txt" and the number 17 or more.

Some sample calls and their results are:

```
>>> main( )
Enter the name of the input file: testFile.txt
Line length should be as long as the longest line to print, or
longer.
Enter the desired line length: 20
```

```
This  is   a   small
test            file
for
you  to  play  with.
The             end.

>>> main( )
Enter the name of the input file: gettysburg.txt
Line length should be as long as the longest line to print, or
longer.
Enter the desired line length: 59

FOUR  SCORE  AND  SEVEN  YEARS  AGO  OUR  FATHERS   BROUGHT
FORTH  ON  THIS  CONTINENT  A  NEW  NATION   CONCEIVED   IN
LIBERTY AND DEDICATED  TO  THE  PROPOSITION  THAT  ALL  MEN
ARE                     CREATED                      EQUAL.

NOW WE ARE ENGAGED IN A GREAT  CIVIL  WAR  TESTING  WHETHER
THAT NATION OR ANY NATION SO CONCEIVED AND SO DEDICATED CAN
LONG ENDURE. WE ARE MET ON A GREAT BATTLEFIELD OF THAT WAR.
WE HAVE COME TO DEDICATE A PORTION OF THAT FIELD AS A FINAL
RESTING PLACE FOR THOSE WHO  HERE  GAVE  THEIR  LIVES  THAT
THAT             NATION             MIGHT             LIVE.
```

To understand how main works, it is necessary to know a little about how to read from a file. The invocation open(file_name, "r") returns a Python file object, attaching the file named by the first argument (file_name) to the file object. The second argument ("r") signifies that the function needs to read data from the attached file. The file is said to be “open for reading” after this invocation.

Once the file is open for reading, its contents can be input a line at a time using a for loop. On the first iteration of the for loop in main, the variable line is assigned a string copy of the first line of the file (sequence of characters starting with the first character and ending with the first newline (‘\n’), inclusive). On the next iteration, variable line is assigned a string copy of the second line, and so on. Unlike when reading input typed in the shell, the newline at the end of a line is included in the string copy.

The remaining operation of main should be easy to understand because of knowing the contracts for the called functions. It should not be necessary to inspect how each of those functions are coded to reason about what the effects of executing main are.

Part 2

Next, you will use PythonTutor to visualize execution of the pad_words function that we supplied. Part 1 illustrated that you don’t need to understand how pad_words works in order to use it in another function—you just need to know its contract so you can call it correctly and know what result it will return. But studying how pad_words works will help you become more familiar with common programming idioms.

For this part, first, load python_tutor_exercise.py into Idle. Replace the (stubbed) helper functions with the functions you wrote in Part 1 and save it. You may want to strip out the doc strings for easier viewing of the executable code in PythonTutor.

Next, bring up a browser and point it to www.pythontutor.com. Watch the "Python Tutor – 1-minute introduction" video, then click "Visualize your code and get live help now" link. Copy and paste the contents of python_tutor_exercise.py into the code window (replacing the sample program in that window). At the top of the code window make sure "Python 3.6" is selected as the language to write code in. Then press the "Visualize Execution" button. Step through the execution (repeatedly press "Forward") until you are confident that you understand how pad_words works.

Call your instructor over and show them a few steps of your visualization and briefly explain the algorithm used in pad_words.

Submitting Your Lab

Put all the Python files you wrote or downloaded for this lab in a folder named `lab-4.py`, then copy it and all its contents into the folder you've shared with your instructor.