Lab 3

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

With this lab we are going to introduce **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

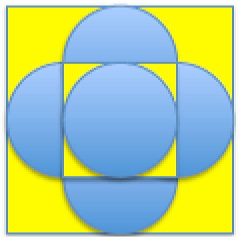
driver and observer roles with only the driver typing and the observer making comments or driving the direction of the code.

Exercise 1: Landscape Design

Programs are good at performing routine mathematical calculations. By way of illustration, you will write a program to calculate the materials needed for an ornamental garden according to the design below. In this design, the blue areas represent flowerbeds and the yellow areas are filled with stone, mulch, or other fill material. The garden is a perfect square. The four outer flowerbeds are congruent semicircles and the central flowerbed is a perfect circle.

Specification

Your program should prompt the user for the following information:



1. The side length (in feet) of the finished garden.
2. The recommended spacing (in feet) between plants.
3. The depth (in feet) of the flowerbeds.
4. The depth (in feet) of the filled areas.

Next estimate the number of plants and the amount of fill and flowerbed soil needed.

Finally, it should report the following quantities needed for the garden:

1. Number of plants for each type of flowerbed (semicircle and circle) and total number of plants for the garden.
2. Cubic yards of soil for each type of flowerbed (semicircle and circle) and total cubic yards of soil for the garden, rounded to one decimal place.
3. Total cubic yards of fill material for the garden, rounded to one decimal place.

Hints

To clarify the problem specifications there is a snapshot of the program's output below.

The formula for the area A of a circle is $A = \pi r^2$, where *r* denotes the radius. For this calculation, you should use the value of π provided in the Python math module.

To estimate the number of plants for a flowerbed, divide the area of the flowerbed by the area needed per plant (the square of the recommended distance between plants) and then truncate this result. Be careful, the number of plants in a semicircle may not be exactly half of the plants in a full circle.

Assume that the user only inputs numbers, that is, you don't need to do error checking for bad input.

Remember to convert fill to cubic yards.

Questions

Answer the following questions in a set of comments in your source file:

1. What happens when the user enters a letter instead of a number at the prompt?
2. What are two ways to get the program to crash (produce an error instead of printing output)?
3. Can you find inputs for which the number of plants required for the center circular flowerbed is not twice that required for a semi-circular bed?

Sample run

```
Calculate Garden requirements
-----------------------------
Enter length of side of garden (feet): 10
Enter spacing between plants (feet): 0.5
Enter depth of garden soil (feet): 0.8333
Enter depth of fill (feet): 0.8333
-----------------------------
Requirements
Plants for each semicircle garden: 39
Plants for the circle garden: 78
Total plants for garden: 234
Soil for each semicircle garden: 0.3 cubic yards
Soil for the circle garden: 0.6 cubic yards
Total soil for the garden: 1.8 cubic yards
Total fill for the garden: 1.3 cubic yards
```

Write and save your code in a file named `garden.py` for full credit.

Exercise 2: Gibberish Game

In many languages, there are games that people play to make normal speech sound incomprehensible, except for a few people who are part of the game. Instead of creating a completely new language, certain sounds are added to words following rules only known to those who are playing, so that anyone else listening will hear only "gibberish" or nonsense words. For some basic examples, see http://en.wikipedia.org/wiki/Gibberish_(language_game).

We are going to create some simple rules for translating normal English into Gibberish. A common rule is to add sounds to each syllable, but since syllables are difficult to detect in a simple program, we'll use a rule of thumb: every vowel denotes a new syllable. Since we are adding a Gibberish syllable to each syllable in the original words, we must look for the vowels.

To make things more unique, we will have two different Gibberish syllables to add. The first Gibberish syllable will be added to the first syllable in every word, and a second Gibberish syllable will be added to each additional syllable. For example, if our two Gibberish syllables were "ib" and "ag", the word "program" would translate to "pribogragam."

In some versions of Gibberish, the added syllable depends on the vowels in a word. For example, if we specify "*b" that means we use the vowel in the word as part of the syllable: e.g. "dog" would become "dobog" (inserting "ob" where the "*" is replaced by the vowel "o") and "cat" would become "cabat" (inserting "ab" where "a" is used). Note that the "*" can only appear at the beginning of the syllable (to make your programming easier).

After the Gibberish syllables are specified, prompt the user for the word to translate. As you process the word, make sure you keep track of two things. First, if the current letter is a vowel, add a Gibberish syllable only if the previous letter was not also a vowel. This rule allows us to approximate syllables: translating "weird" with the Gibberish syllable "ib" should become "wibeird", not "wibeibird". Second, if we've already added a Gibberish syllable to the current word, add the secondary syllable to the remaining vowels. How can you use Booleans to handle these rules?

Finally, print the Gibberish word. Afterwards, ask the user if they want to play again, and make sure their response is an acceptable answer (“yes”/“no”, “y”/“n”)

Specification

Your program will:

1. Print a message explaining the game.
2. Prompt for two Gibberish syllables (indicate the allowed wildcard character “*”).
3. Prompt for a word to translate.
4. Process the word and add the syllables where appropriate.
5. Print the final word, and ask if the user wants to play again.

Hints

You should start with this program, as with all programs, by breaking the program down into parts.

1. Getting started: Simplify!
   a) First solve the program using a single Gibberish syllable, without checking for multiple vowels in a row or using the wildcard (“*”). This means that “weird” with the “ib” syllable will become “wibeibird” (that will not be correct in the final version, but good enough for starting your program: simplify!).
   b) You will need to decide how you will check for vowels. Good possibilities are string indexing or using a Boolean, but use a method that works best with your own program. When you check for vowels it may be handy to create a string like `vowels = “aeiouAEIOU”` and use `in vowels` to check if a character is a vowel (is the character in the string named vowels).
   c) In this simplified version assume that the Gibberish syllable is exactly two characters long.
   d) For your resulting word start with an empty string and add characters onto it as you process characters from the original word.
2. After that, add the second Gibberish syllable and allow Gibberish syllables longer than two characters.
   a) A Boolean to keep track of whether you have already made a substitution for the first Gibberish syllable will be useful, e.g. `done_with_first_vowel = False`
   b) Slicing may be useful for longer Gibberish syllables.
3. Add the wildcard ability after you’ve completed the above steps.
4. Finally, add handling of the special case of consecutive vowels. For most people this is the hardest part of the program. It doesn’t add much Python code, but until you see it the logic can be elusive.
5. The string library has a couple of useful tools. If you add import string at the beginning of your program, `string.digits` and `string.ascii_letters` are strings that contain all the digits (0 through 9) and all the letters (uppercase and lowercase).

Sample run

```
Enter your first Gibberish syllable (add * for the vowel substitute): i3

Syllable must only contain letters or a wildcard ('*'): ip
```

```
Enter the second Gibberish syllable (* for vowel substitute): *zz

Please enter a word you want to translate:

--> Gibberish

Your final word:

Gipibbezzerizzish

Play again? (y/n) m

Please enter y to continue or n to quit: n

Thanks for playing!
```

Write and save your code in a file named `gibberish.py` for full credit.

Submitting Your Lab

Put all the Python files you wrote for this lab in a folder named `lab-3`, then copy it and all its contents into the folder you've shared with your instructor.