CSC1015F Assignment 6

Functions

Assignment Instructions

Previous assignments have involved solving programming problems using input and output statements, 'if' and 'if-else' control flow statements, 'while' statements, 'for' statements, and statements that perform numerical and string manipulation.

This assignment builds on these technologies and offers practice using functions and modules. Functions are very effective when used in conjunction with a divide-and-conquer approach to problem solving. An example on the use of functions is given in **Appendix A.**

NOTE Your solutions to this assignment will be evaluated for correctness and for the following qualities:

- Documentation
  - Use of comments at the top of your code to identify program purpose, author and date.
  - Use of comments within your code to explain each non-obvious functional unit of code.
- General style/readability
  - The use of meaningful names for variables and functions.
- Algorithmic qualities
  - Efficiency, simplicity

These criteria will be manually assessed by a tutor and commented upon. In this assignment, up to 10 marks will be deducted for deficiencies.

Question 1 [30 marks]

Write a Python module called `boxes.py` that contains the following 3 functions to create hollow boxes of stars.

- `print_square ()`, that returns a string containing a 5x5 box.
- `print_rectangle (width, height)`, that prints a box on the screen with given width and height
- `get_rectangle (width, height)`, that returns a string containing a box with given width and height

A python module is a file containing Python code just like a regular program. Typically such a module will contain only functions. To use the module in another program, we first issue the import statement, then refer to each function by prefixing it with the module name.

The Vula page for this assignment has the main program called ‘`drawings.py`’. Download this program and use it to test your program and **DO NOT** change this file.

Here is a program skeleton:

```
import math
def print_square():
      # Your code here
def print_rectangle(width, height):
      # Your code here
def get_rectangle(width, height):
      # Your code here
```

Sample I/O (**Note**: *the input from the user has been shown in **bold***):

```
Choose test:
a
* * * * *
*       *
*       *
*       *
* * * * *
```

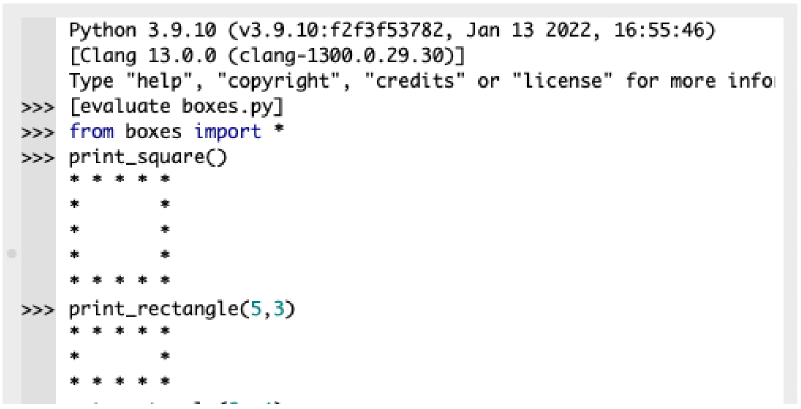
Sample I/O (**Note**: *the input from the user has been shown in **bold***):

```
Choose test:
b 3 4
calling function
* * *
*   *
*   *
* * *
called function
```

Sample I/O (**Note**: *the input from the user has been shown in **bold***):

```
Choose test:
c 4 3
calling function
called function
****
*  *
****
```

Using the Wing IDE Python shell, you should be able to test your functions as illustrated with the `side.py` program example given in **Appendix A**. For example,



```
    Python 3.9.10 (v3.9.10:f2f3f53782, Jan 13 2022, 16:55:46)
    [Clang 13.0.0 (clang-1300.0.29.30)]
    Type "help", "copyright", "credits" or "license" for more info
>>> [evaluate boxes.py]
>>> from boxes import *
>>> print_square()
    * * * * *
    *       *
    *       *
    *       *
    * * * * *
>>> print_rectangle(5,3)
    * * * * *
    *       *
    * * * * *
```

NOTE that the automatic marker will test each of your functions individually. To enable this, your program MUST, as shown in the skeleton, have the following lines at the end:

```
if __name__=='__main__':
    main()
```

The way in which functions are tested is like that illustrated with the Wing IDE Python shell (see the appendix at the end of this assignment sheet). A trial consists of trying to execute a code snippet that uses the function under test. If a trial fails, typically, you will see the code snippet – an import statement followed by one or more function call expressions.

Question 2 [20 marks]

Computers often interface with the outside world using external sensors and actuators. For example, weather systems have sensors to detect the temperature and pressure. These sensors usually have limited computational ability, so they produce data in raw form and this data needs to be captured and processed by a program. On the Vula assignment page, you will find a skeleton for a program called ‘`extract.py`’. The intent is that the program be used to extract useful data from a raw data stream obtained from a sensor.

The data from the sensor contains a block with the format:

... `BEGIN temp_press:Xcoordinate,Ycoordinate emanetiS END` ...

The program must output the data in the following format:

```
Location: Sitename
Coordinates: Ycoordinate Xcoordinate
Temperature: temp C
Pressure: press hPa
```

Sample I/O (**Note**: *the input from the user has been shown in **bold***):

```
Enter the raw data:
fbkjf kfjkdb fds BEGIN 12.2_1014:18.6E,34.0S NWOT EPAC END fdfdfds
Site information:
Location: Cape Town
```

```
Coordinates: 34.0S 18.6E
Temperature: 12.20 C
Pressure: 1014.00 hPa
```

To complete the program you must implement the *location*, *temperature*, *pressure*, *y_coordinate*, *x_coordinate*, and *get_block* functions. The functions have been identified by applying a divide-and-conquer strategy. Each solves a part of the overall problem:

- `get_block(raw_data)`

  Given a string of raw data as a parameter, the *get_block* function extracts the sub string starting with 'BEGIN' and ending with 'END'.
- `location(block)`

  Given a block string as a parameter, the *location* function returns the location component in title case.
- `pressure(block)`

  Given a block string as a parameter, the *pressure* function returns the pressure component as a real number value.
- `temperature(block)`

  Given a block string as a parameter, the *temperature* function returns the temperature component as a real number value.
- `y_coordinate(block)`

  Given a block string as a parameter, the *y_coordinate* function returns the y coordinate component as a string.
- `x_coordinate(block)`

  Given a block string as a parameter, the *x_coordinate* function returns the x coordinate component as a string.

Examples

- `get_block('fds BEGIN 12.2_1014:18.6E,34.0S NWOT EPAC END fdf fds ')` returns the block string `'BEGIN 12.2_1014:18.6E,34.0S NWOT EPAC END'`.
- `location('BEGIN 12.2_1014:18.6E,34.0S NWOT EPAC END')` returns `'Cape Town'` .
- `pressure('BEGIN 12.2_1014:18.6E,34.0S NWOT EPAC END')` returns 1014.0.
- `temperature('BEGIN 12.2_1014:18.6E,34.0S NWOT EPAC END')` returns 12.2.
- `y_coordinate('BEGIN 12.2_1014:18.6E,34.0S NWOT EPAC END')` returns `'34.0S'`.
- `x_coordinate('BEGIN 12.2_1014:18.6E,34.0S NWOT EPAC END')` returns `'18.6E'`.

NOTE: The automatic marker will test each of your functions individually. To enable this, you MUST NOT remove the following lines from the skeleton:

```
if __name__=='__main__':
       main()
```

Question 3 [20 marks]

Programs are often refactored or reorganised to make better use of functions and thereby reduce redundancy and improve reusability of code.

You have been provided with the complete `combine.py` Python program that calculates the number of *k*-permutations of *n* items. A much more readable and space-efficient program is also provided as `fcombine.py`. You must reuse/adapt the code in the complete program to create the `mymath.py` module with the following 2 required functions:

- `get_integer(s)` accepts an integer, *s*, from the user.
- `calc_factorial(n)` iteratively calculates the factorial of an integer *n*.

The Vula page for this assignment has the programs combine.py and fcombine.py. Download these programs and use them to test your program.

Sample I/O (**Note**: *the input from the user has been shown in **bold***):

```
Enter n:
4
Enter k:
2
Number of permutations: 12
```

Question 4 [30 marks]

This question concerns completing a skeleton for a program called *calendar_month.py* that accepts the name of a month and a year as input and prints out the calendar for that month. Download this program from the Vula page for this assignment.

Sample I/O (**Note**: *the input from the user has been shown in **bold***):

```
Enter month:
May
Enter year:
2020
May
Mo Tu We Th Fr Sa Su
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
```

To complete the program you are required to complete a number of functions, including a *main* function.

- `day_of_week(day, month, year)`
  Given a date consisting of day of month, month number and year, return the day of the week on which it falls. The function returns 1 for Monday, 2 for Tuesday, ..., 7 for Sunday.
- `is_leap(year)`
  Given a year return `True` if it is a leap year, `False` otherwise. This function returns a *Boolean* value.
- `month_num(month_name)`
  Given the name of a month, return the month number i.e. 1 for January, 2 for February, ..., 12 for December. The name can be in UPPER CASE, lower case or Title Case.
- `num_days_in(month_num, year)`
  Given a month number and year, return the number of days in the month.
- `num_weeks(month_num, year)`
  Given a month number and year, return the number of weeks that the month spans. (The first week is the week in which the first of the month falls, the last week is the week in which the last day of the month falls. Counting from first to last gives the number of weeks.)
- `week(week_num, start_day, days_in_month)`
  Given a week number, (1st, 2nd, ...), the day on which the 1st of the month falls (1 for Monday, 2 for Tuesday, ...), and the number of days in the month, return a string consisting of the day of the month for each day in that week, starting with Monday and ending with Sunday.
- `main()`
  Obtain the name of a month and a year from the user and then print the calendar for that month by obtaining the number of weeks and then obtaining the week string for each.

The functions have been identified by applying a divide-and-conquer strategy. Each solves a part of the overall problem.

Examples

- `day_of_week(1, 4, 2020)` returns integer 3 (which represents Wednesday).
- `is_leap(2020)` returns the Boolean value True.
- `month_num('April')` returns integer 4.
- `num_days_in(4, 2020)` returns integer 30.
- `num_weeks(4, 2020)` returns integer 5.
- `week(1, 3, 30)` returns the string `'      1  2  3  4  5'`
- `week(2, 3, 30)` returns the string `' 6  7  8  9 10 11 12'`

HINTS:

- Days of the month.
- We can use Gauss's formula to calculate the day of the week on which the 1st of January of a given year falls. This could be adapted for the *day_of_week* function.

  Alternatively, you could consider Zeller's congruence:

  Given day of the month, *d*, month number, *m*, and year, *y*;

  If the month is January or February, then add 12 to *m* and subtract 1 from y.

  Compute:

$$\left(d + \left\lfloor\frac{13(m + 1)}{5}\right\rfloor + y + \left\lfloor\frac{y}{4}\right\rfloor - \left\lfloor\frac{y}{100}\right\rfloor + \left\lfloor\frac{y}{400}\right\rfloor\right) \mathit{modulus}\ 7$$

  The result, *h*, is a number 0 to 6 where 0 is Saturday, 1 is Sunday, …, and 6 is Friday.

  The formula $(h + 5)$ **modulus** $7) + 1$ will give a value from 1 to 7 where 1 is Monday, 2 is Tuesday, …, and 7 is Sunday.
- The floor of a value, *v*, written $\lfloor v \rfloor$, is the largest integer value that is smaller than *v*. In Python you write '`math.floor(v)`'.

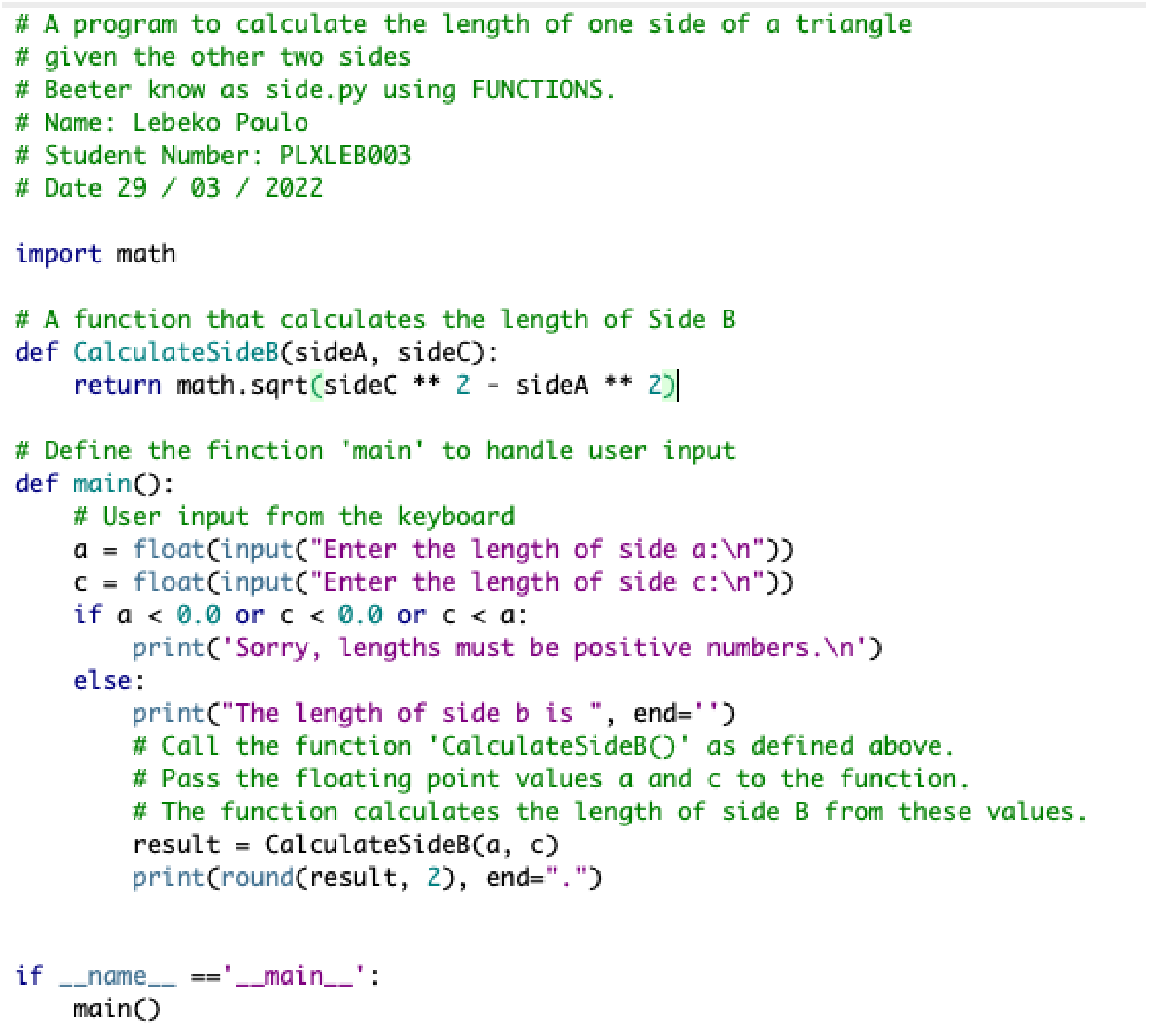
NOTE: The automatic marker will test each of your functions individually. To enable this, you MUST NOT remove the following lines from the skeleton:

```
if __name__=='__main__':
    main()
```

END

APPENDIX A (based on Assignment 2 – Question 4)

To explain by example, Assignment 2 Question 4 asks you to write a program called `side.py` to calculate the side, *b,* of a right-angled triangle given the sides *a* and *c* using Pythagoras' Theorem. Here is a solution that demonstrates the use of functions:



```python
# A program to calculate the length of one side of a triangle
# given the other two sides
# Beeter know as side.py using FUNCTIONS.
# Name: Lebeko Poulo
# Student Number: PLXLEB003
# Date 29 / 03 / 2022

import math

# A function that calculates the length of Side B
def CalculateSideB(sideA, sideC):
    return math.sqrt(sideC ** 2 - sideA ** 2)

# Define the finction 'main' to handle user input
def main():
    # User input from the keyboard
    a = float(input("Enter the length of side a:\n"))
    c = float(input("Enter the length of side c:\n"))
    if a < 0.0 or c < 0.0 or c < a:
        print('Sorry, lengths must be positive numbers.\n')
    else:
        print("The length of side b is ", end='')
        # Call the function 'CalculateSideB()' as defined above.
        # Pass the floating point values a and c to the function.
        # The function calculates the length of side B from these values.
        result = CalculateSideB(a, c)
        print(round(result, 2), end=".")


if __name__ =='__main__':
    main()
```

There are two functions, one called '`main`', and the other '`CalculateSideB`'. Between them, they break the problem into two parts.

- The *main* function is responsible for handling user input and output.
- The *CalculateSideB* function is responsible for calculation.

The *main* function contains a 'function call'.

- It calls *CalculateSideB*, passing the floating point values of *a* and *c*.
- The *CalculateSideB* function uses the values to calculate and return the value of the side *b*.
- The *main* function assigns the value it receives to the variable '*result*'.
- On the next line (and final line) it prints out the value of *result* correct to 2 decimal places.

By breaking the programming problem into two parts, each can be concentrated on without concern for the other. The *CalculateSideB* function can be developed without concern for where a and c come from. The *main* function can be developed without concern for exactly what *CalculateSideB* will do. It's enough to know simply that it requires two values (the sides *a* and *c*) and that it will calculate the length of *b*.

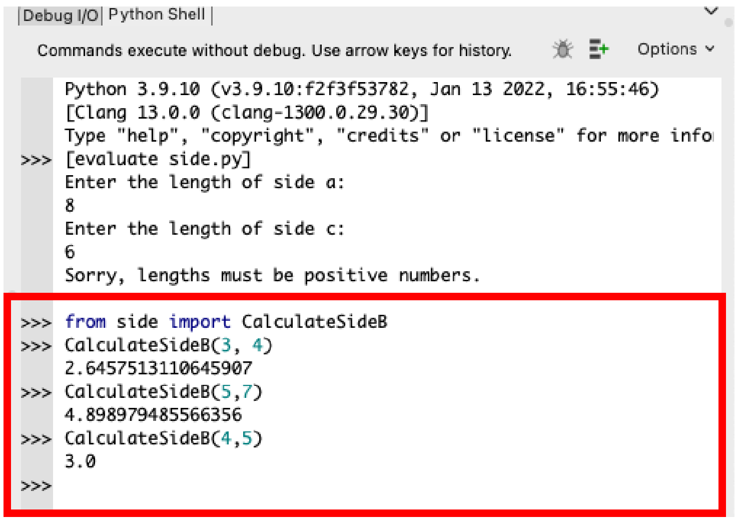
Admittedly, this programming problem is simple, and you probably solved it quite satisfactorily when working on Assignment 2, however, it serves to convey the idea.

The idea of divide-and-conquer works best if you have the techniques and technology to fully support working on one part without concern for another. If, say, you were developing the Pythagoras' Theorem program and you chose to concentrate on the *CalculateSideB* function first, you'd probably want to check it worked correctly. But surely that means you need the *main* function so that you can obtain useful inputs?

A technique for dealing with this is to have a '`stub`' *main* function which with which to make test calls to *CalculateSideB* e.g.

```
def main():
      print(CalculateSideB(3, 4))
      print(CalculateSideB(5, 7))
      # ...
```

Alternatively, Wing IDE provides a piece of technology in the form of a Python shell. Within the shell you can import functions that you are working on and then write expressions that use them (see the function calls inside the red rectangle below):



```
Python 3.9.10 (v3.9.10:f2f3f53782, Jan 13 2022, 16:55:46)
[Clang 13.0.0 (clang-1300.0.29.30)]
Type "help", "copyright", "credits" or "license" for more info
>>> [evaluate side.py]
Enter the length of side a:
8
Enter the length of side c:
6
Sorry, lengths must be positive numbers.
>>> from side import CalculateSideB
>>> CalculateSideB(3, 4)
2.6457513110645907
>>> CalculateSideB(5,7)
4.898979485566356
>>> CalculateSideB(4,5)
3.0
>>>
```

The screenshot illustrates its use. Lines entered by the user have a prompt, '>>>' beside them. Lines without the prompt are responses from the Python shell.

1. The user enters an `import` statement for the *CalculateSideB* function.

2. The user then enters a function call expression, calling *CalculateSideB* with the values 3 and 4 for *a* and *c*.

3. The result, 2.6457513110645907 is printed on the next line.

4. The user enters another function call expression, this time with the values 5, and 7,

5. And the result, 4.898979485566356 is printed on the next line.

etc….