COMP 1510 Assignment #3: Functions, functions, functions!

Christopher Thompson
chris_thompson@bcit.ca

Due **Sunday February 13th at or before 12:00:00 noon**

Welcome!

The fundamental building block of Python is the statement or command, which we assemble into named, re-usable blocks of code called functions.

We assemble our code into functions to make our code modular, i.e., easier to write, use, maintain, and grow. Writing short, atomic functions means we can guarantee that each function does exactly what it must through the use of some accompanying doctests.

For your third assignment, I challenge you to implement and test the functions described below. Good luck, and have fun!

1 Submission Requirements

1. This assignment is due no later than **Sunday February 13th at or before 12:00:00 noon.**
2. **Late submissions will not be accepted for any reason.**
3. **This is an individual assignment**. I strongly encourage you to share ideas and concepts, but sharing code or submitting another person's code is not allowed.

2 Project Setup

1. Follow this link to accept the assignment from GitHub Classroom, and copy the PyCharm template I created to your personal list of repositories:

   `https://classroom.github.com/a/LaFNc1wP`

2. Clone the project to your local machine.
3. Add your name and student number to the README.md.
4. Commit and push. Hello world!

3 Style Requirements

1. You must observe the code style warnings produced by PyCharm. **You must style your code so that it does not generate any warnings.**

   When code is underlined in red, PyCharm is telling us there is an error. Mouse over the error to (hopefully!) learn more.

When code is underlined in a different colour, we are being warned about something. We can click the warning triangle in the upper right hand corner of the editor window, or we can mouse over the warning to learn more.

2. **You must comment each function you implement with correctly formatted docstrings.**
3. **Include informative doctests where necessary and possible** to demonstrate to the user how the function is meant to be used, and what the expected reaction will be when input is provided.
4. For this assignment, functions must only do one logical thing. If a function does more than one logical thing, break it down into two or more functions that work together. Remember that helper functions may help more than one function, so we prefer to use generic names as much as possible. Don't hard code values, either. Make your functions as flexible as possible.
5. Don't create functions that just wrap around and rename an existing function, i.e., don't create a divide function that just divides two numbers because we already have an operator that does that called / .
6. **Ensure that the docstring for each function you write has the following components (in this order)**:
   (a) Short one-sentence description that begins with a verb in imperative tense
   (b) One blank line
   (c) Additional comments if the single line description is not sufficient (this is a good place to describe in detail how the function is meant to be used)
   (d) One blank line
   (e) PARAM each parameter needs its own PARAM tag which describes what the user should pass to the function.
   (f) PRECONDITION statement for each precondition which the user promises to meet before using the function
   (g) POSTCONDITION statement for each postcondition which the function promises to meet if the precondition is met
   (h) RETURN statement which describes what will be returned from the function if the preconditions are met
   (i) One blank line
   (j) **And finally, yes, include helpful doctests.**
   (k) Here is an example:

```
def my_factorial(number):
    """Calculate factorial.

    A simple function that demonstrates good comment construction.

    :param number: a positive integer
    :precondition: number must be a positive integer equal to or
                   greater than zero
    :postcondition: calculates the correct factorial
    :return: factorial of number

    >>> my_factorial(0)
    1
    >>> my_factorial(1)
    1
    >>> my_factorial(5)
    120
    """
    return math.factorial(number)
```

4 Flowcharts and Computational Thinking

In the next section there is a list of ten functions you must implement. For each function:

1. **Create a flowchart illustrating the function's logic.** You may only use the standard ANSI flowchart control structures we have discussed in class.

2. **Each flowchart must be in its own PDF file** with the same name as the module that contains the function described in the flowchart.

3. For each of the functions below, **add up to four statements to the README.md that describe how you used some or all of the four components of computational thinking to design your solution**. Recall that the four components of computational thinking are decomposition, pattern matching/data representation, abstraction/generalization, and algorithms/automation. This means that for each function you will have four or fewer sentences in the README.md. This can be an organic process and we often use these tools without even realizing it! Be mindful of where your thoughts and ideas go.

5 Functions

Please implement the following ten functions. For some functions, you may wish to (and perhaps even should) create helper functions:

1. **Implement a function called seconds in a file called seconds.py**. Create a flowchart in a file called seconds.PDF. The function accepts integer parameters **in this order** for weeks, days, hours, and minutes. Some or all of these values may be negative integers.

   This function must return the total number of seconds represented by the arguments. Positive arguments should be added to the total, and negative arguments should be deducted from the total. Return an int, not a float. You may assume that there are:

   (a) 60 seconds in 1 minute

   (b) 3,600 seconds in 1 hour

   (c) 86,400 seconds in 1 day

   (d) 7 days in 1 week.

   Return the number of seconds as an integer. Yes, it could be a negative integer. Do not print anything!

2. Long long ago, in a galaxy far, far away, interest rates were so good that a simple savings account earned 10% interest a year. Yes, ten percent! Those heady days (the 80s) are long over, and today the miracle of compound interest is less of a miracle and more a footnote of history. But let's pretend...

   When a bank pays compound interest, the interest is earned not only on the principal but also the interest that has already accumulated over time. If we deposit some money into an account, and let the account earn compound interest over a certain number of years, the formula for calculating the balance after a certain number of years is:

$$A = P(1 + \frac{r}{n})^{nt}$$

   Where:

   (a) A is the amount of money in the account after the specified number of years

   (b) P is the principal amount that was originally deposited into the account

   (c) r is the annual interest rate

   (d) n is the number of times per year that the interest is compounded

   (e) t is the specified number of years.

Implement a function called compoundinterest in a file called compoundinterest.py. Create a flowchart in a file called compoundinterest.PDF. The function must accept the following parameters in this order:

(a) principal (a float that may be positive or negative)

(b) annual interest rate earned by the account (a float that may be positive or negative, where 0.1 is equal to 10 percent interest per year)

(c) the number of times per year the interest is compounded (an int that is greater than zero)

(d) the number of years the account will be left alone to grow (a float that is greater than 0.

Return A, the amount of money in the account after the elapsed time, as a float. Use better variable names than me, though! And don't print anything!

3. **Implement a function called loshumagic in a file called loshumagic.py**. Create a flowchart in a file called loshumagic.PDF.

   The Lo Shu Magic Square is a grid with three rows and three columns. The Lo Shy Magic Square has the following properties:

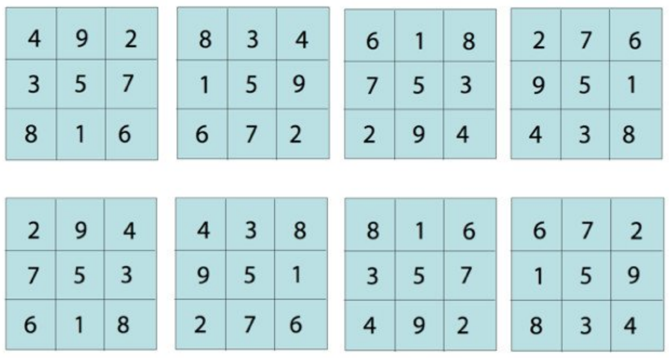
   (a) The grid contains three rows and three columns

   (b) The grid contains the numbers 1 through 9 (no duplicates!)

   (c) The sum of each row, each column, and each diagonal all add up to the same number.

   In Python we can model a magic square using a 2-dimensional list.

   Implement a function called loshumagic that accepts a 2-dimensional list as an argument and determines whether the list is a Lo Shu Magic Square. If so, return True, else return False.



4	9	2
3	5	7
8	1	6

8	3	4
1	5	9
6	7	2

6	1	8
7	5	3
2	9	4

2	7	6
9	5	1
4	3	8

2	9	4
7	5	3
6	1	8

4	3	8
9	5	1
2	7	6

8	1	6
3	5	7
4	9	2

6	7	2
1	5	9
8	3	4

Figure 1: *Some Lo Shu Magic Squares (wow!)*

4. **Implement a function called romannumeral in a module called romannumeral.py**. Create a flowchart in a file called romannumeral.PDF. The function must accept a single parameter called positive_int, and return the Roman numeral which is equivalent. Return the Roman numeral as a string that does not contain any leading, inner, or trailing whitespace. Do not print anything!

   Your function is only responsible for providing a correct answer when the input is a positive integer in the range [1, 1000]. You are not responsible for the behaviour of your function if the user passes any other argument.

   The Roman number system was used in Europe until the Middle Ages. Numbers are represented by combinations of letters from the Roman alphabet.

   We will use a subset of the Roman number system, to wit we will only use the following symbols:

   (a) I (the capital i) represents 1

(b) V represents 5
(c) X represents 10
(d) L represents 50
(e) C represents 100
(f) D represents 500
(g) M represents 1000.

5. **Implement a function called phone in a file called phone.py**. Create a flowchart in a file called phone.PDF. Many companies like restaurants use phone numbers like 555-PHO-KING (that pho's not just good, it's Pho King good!), so the number is easier for their customers to remember. On a standard phone, the alphabetic numbers are mapped to digits in the following fashion:

   (a) A, B, and C = 2
   (b) D, E. and F = 3
   (c) G, H, and I = 4
   (d) J, K, and L = 5
   (e) M, N, and O = 6
   (f) P, Q, R, and S = 7
   (g) T, U, and V = 8
   (h) W, X, Y, and Z = 9

   Your function must accept a string. Your function will only work for strings that are provided in the format XXX-XXX-XXXX where each X is alphanumeric. If your function receives a string that is not formatted like this, your function probably won't work. That is correct. Your function must return the telephone number with any alphabetical numbers translated into their numerical equivalent. For example, if the string provided is 555-GET-FOOD, the function must return the string "555-438-3663". Don't print anything!

6. **Implement a function called leapyears in a file called leapyears.py** Create a flowchart in a file called leapyears.PDF:

   (a) Visit the Python documentation page for the calendar module at `https://docs.python.org/3/library/calendar.html` and examine the functions that are available. You may find one or more that you can use inside the function you write.
   (b) Your function called leapyears must accept two parameters, lower_bound and upper_bound. Your function is only responsible for working if lower_bound is less than or equal to upper_bound and both values are integers greater than zero.
   (c) Your function must calculate how many leap years there will be in the range [lower_bound, upper_bound].
   (d) Return the number of leap years in the range [lower_bound, upper_bound] as an integer. Don't print anything!

7. Eratosthenes (Air-uh-TOSS-the-knees) was a Greek mathematician who developed an efficient method for identifying prime numbers. A prime number is a number greater than 1 that is only divisible by 1 and itself, i.e. 7 is prime because it is only divisible by 1 and 7, but 8 is not prime because it is divisible by 1, 2, 4, and 8.

   His method, called the Sieve of Eratosthenes, works like this:

   (a) Make a list of numbers: 0, 1, 2, 3, ..., n.
   (b) 0 is not prime, so cross it out.
   (c) 1 is not prime, so cross it out.
   (d) 2 is prime, but its multiples are not prime, so cross out 4, 6, 8, 10, ...
   (e) 3 is prime, but its multiples are not prime, so cross out 6, 9, 12, 15, ...

(f) (4 is crossed out so next is 5) 5 is prime, but its multiples are not primes, so cross out 10, 15, 20, 25, ...

(g) Continue until you have reached $\sqrt{n}$. (can you explain why it is sufficient to stop here?) What has not been crossed out is a prime number!

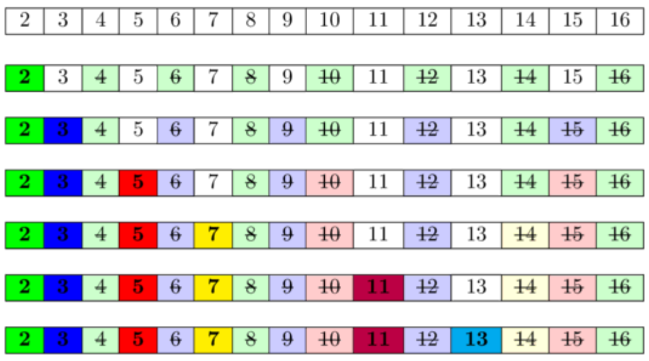


Figure 2: *The sieve algorithm in colour...*

(h) **Implement the sieve of Eratosthenes in a function called eratosthenes in a file called eratosthenes.py.** Create a flowchart in a file called eratosthenes.PDF. This function will accept a single positive integer called upper_bound. If the value passed to the eratosthenes function is not a positive integer, your function is not responsible for what happens.

(i) Use the Sieve of Eratosthenes to determine which of the numbers in the range [0, upper_bound] are prime numbers.

(j) Return the primes between [0, upper_bound] as a list. For example:

```
primes_below_30 = eratosthenes(30)
print(primes_below_30)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

8. **Implement a function called moneychanger in a file called moneychanger.py.** Create a flowchart in a file called moneychanger.PDF.

   (a) The moneychanger function accepts a floating point number that represents an amount of Canadian money, and determines the fewest of each bill and coin we need to represent it.

   (b) We will consider the following denominations: 100, 50, 20, 10, 5, 2, 1, 0.25, 0.10, 0.05. There are no pennies. You must round the value correctly.

   (c) If the value entered is not a positive float that only has 2 decimal places, your function is not responsible for how it behaves.

   (d) The function must return a list that tells me how many of each denomination are required. For example, if 66.53 is entered, the program must return a list that looks like this:

```
breakdown = cash_money(66.53)
print(breakdown)
[0, 1, 0, 1, 1, 0, 1, 2, 0, 1]
```

   (e) Be careful! There may be one or more values that seem... tricky. This is a consequence of using 2s complement and binary. How can we avoid this situation?

9. **Implement a function called caesarcipher in a file called caesarcipher.py.** Create one (or more!) flowcharts in a file called caesarcipher.PDF.

   (a) The Caesar cipher is one of the earliest known and simplest ciphers. It is a type of substitution cipher in which each letter in a plaintext message is 'shifted' a certain number of places down the alphabet. For example, with a shift of 1, A would be replaced by B, B would become C, and so on.

(b) Implement a function called caesarcipher. This function accepts three parameters: a string called message, a Boolean called encode, and an integer representing the shift. The string may be empty. The Boolean must be True or False. The integer representing the shift may be positive or negative.

(c) The caesarcipher function must shift each alphanumeric character in the message the specified number of places. Do not shift characters that are not letters [a, z], [A, Z] or digits [0, 9]. Letters must only consider letters when shifting, i.e., shifting a Z two places in the positive direction replaces the Z with a B. Shifting an a three places in the negative direction replaces the a with an x. Numbers must only consider numbers when shifting, i.e., shifting 0 nine places in the positive direction replaces the 0 with a 9, and shifting 0 ten places in the positive direction replaces the 0 with a 0, and shifting 0 eleven places in the positive direction replaces the 0 with a 1.

(d) The function must return the string's cipher.

(e) Note that the opposite must be done if the Boolean called encode is set to False. If it is set to False, you must decode the cipher. Shift the letters and digits appropriately to discover and return the hidden message.

6 Grading

Your third assignment will be marked out of 15. I will randomly select three (3) of the functions from this assignment and mark them. Don't gamble. Complete all of them. For full marks, you must:

1. **(3) Correctly implement the functional requirements** described in this document for each function I mark (1 mark per function).

2. **(3) Adequately test each function**. Nearly every function and helper function can be tested with doctests to unambiguously demonstrate how the function works. Only test input that meets the precondition, or input that requires an error message. Please note that you cannot write doctests for functions that use random numbers or functions that require user input because I haven't shown you how to do this yet. Clever decomposition of the problems can modularize some or all of the important logic into separate testable helper functions.

3. **(3) Correctly represent the logic for each function in a flowchart** submitted as a PDF (1 mark per function).

4. **(3) Adequately describe the elements of computational thinking** you used when implementing each function. This information must be in your README.md and it must be clearly labeled, tidy, and easy to read. I mark (1 mark per function).

5. **(3) Correctly format and comment your code** for each function I mark (1 mark per function). Eliminate all style warnings offered by PyCharm, use good function and variable names, write code that is easy to understand, use whitespace wisely, employ good grammar in your comments, use inline comments (comments that start with #) inside functions to describe complicated code blocks, make sure functions are brief and only do one thing, etc.

Please remember that this is an individual assignment. I strongly encourage you to share ideas and concepts (please do this!), but sharing code or screen views or submitting someone else's work is not allowed.

Good luck, and have fun!