# COMP 1510 Object oriented programming 1
# Assignment #1: Functions, functions, functions!

Christopher Thompson
`cthompson98@bcit.ca`

Due Saturday September 28th at or before 17:59:59

## 1 Introduction

The first few weeks of CST can be a bit of a blur. There is a lot to process: a new program, new classmates, seven courses (well, six, I think COMP 1100 is pretty easy), and the famous BCIT workload. In 1510, we took a deep breath and dove in. We are well into our exploration of the fundamentals of programming using Python, a very popular multi-paradigm high level language.

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 easier to write and maintain. Writing short, atomic functions means we can guarantee that each function does exactly what it must through the use of accompanying doctests and unit tests.

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

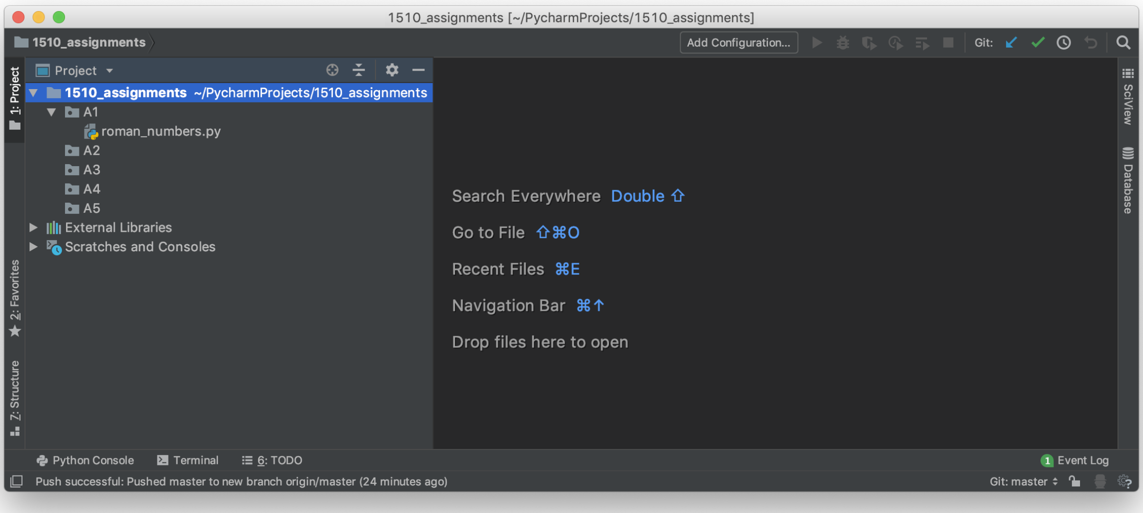
## 2 Submission Requirements

1. **This take home assignment is due no later than Saturday September 28th at 17:59:59.**
2. Late assignments 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 someone else's work is not allowed.

## 3 Project Setup

Please complete the following:

1. Create a new project in PyCharm called A########_1510_assignments, where A######## is your student number. We will use this project for ALL of our assignments this term.
2. Inside the new project, create five Python packages called A1, A2, A3, A4, and A5 respectively. We will use this project for all our assignments this term. For your first assignment, all code must go in the directory called A1.
3. Inside the A1 directory, create a new Python file called roman_numbers.py. When you are done, your project structure should look like Figure 1 on the next page.
4. Add this project to git. From the main menu select VCS > Import into version control > Create git repository. Select the project folder (A########_1510_assignments, not A1).
5. Now add the project to GitHub in the cloud. Select VCS > Import into version control > Share project on GitHub. Ensure the project is private. I will not mark any repository which is public or which has ever been public.



Push successful: Pushed master to new branch origin/master (24 minutes ago)

Figure 1: Your project folders should look like this.

6. For your first assignment, all files must go into the A1 folder. After you complete each function, commit and push your change to version control. In order to earn full marks, you must commit and push after each function is complete.

7. When you are finished, invite me as a collaborator. You will only need to do this for A1. For future assignments, since I am already a collaborator I will pull your future work automatically.

## 4 Style Requirements

1. You must comment each function you implement with correctly formatted docstrings. Include informative doctests **where necessary** to demonstrate to the user how the function is meant to be used, and what the expected output will be when input is provided.

2. Functions must be short and must only do one thing. If a function does more than one thing, break it down (decompose it) into two or more functions that work together. Remember that helper functions can help more than one function, so we want to use generic names as much as possible.

3. Each function must also be tested in a separate unit test file by a separate collection of unit tests. Ensure your test cases and functions have descriptive, meaningful names, and ensure that every possible valid input is tested for each function. Remember we want to create disjointed equivalency partitions out of all possible output, and test a representative from each partition.

4. 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) PARAM statement for each parameter that describes what it should be, and what it is used for

   (e) PRECONDITION statement for each precondition which the user promises to meet before using the function

   (f) POSTCONDITION statement for each postcondition which the function promises to meet if the precondition is met

   (g) RETURN statement that describes what the function returns

(h) One blank line

(i) And finally, the doctests, **if the function needs them**. 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
    :postcondition: calculates the correct factorial
    :return: correctly calculated factorial as an int
    >>> my_factorial(0)
    1
    >>> my_factorial(1)
    1
    >>> my_factorial(5)
    120
    """
    return math.factorial(number)
```

## 5 Functions

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

1. For each of the seven functions described below, use MS Visio (available for free through Azure Dev Tools for Teaching [ADTT] via https://kb.bcit.ca/sr/software/licensing/studentmain.html ) or an online flowchart maker to create a flowchart illustrating the function's logic using the standard ANSI flowchart control structures. Each flowchart must be in its own PDF file with the same name as the file that contains the function described in the flowchart. The PDF file must be dragged and dropped into the correct folder in your PyCharm project. For example, there must be a roman_numbers.pdf file which contains a flowchart describing the logic in your convert_to_roman_numeral function.

2. For each of the seven functions below, add a multi-line comment to the bottom of the source file that describes the components of computational thinking you used to solve the problem. Recall that the four components of computational thinking are decomposition, pattern matching/data representation, abstraction/generalization, and algorithms/automation.

3. Inside the roman_numbers.py file, implement a function called convert_to_roman_numeral. The function should accept a single parameter called positive_int, and return the Roman numeral which is equivalent. Return the Roman numeral as a string.

   Your function is only responsible for providing a correct answer when the input is a positive integer in the range [1, 10_000]. Your function is not required to do anything if it receives anything else.

   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:

   (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.

4. Create a new file called colour_mixer.py inside the A1 directory. In it, implement a function called colour_mixer. This function must prompt the user for two primary colours. The three primary colours are red, yellow, and blue. When we mix two primary colours, we generate a secondary colour:

   (a) mixing red and blue produces purple
   (b) mixing red and yellow produces orange
   (c) mixing yellow and blue produces green (one of my favourites).

   Your function must print a helpful error message if the user enters the two same colours, or colours that are not primary. Otherwise, print the name of the secondary colour that results.

5. Create a new file called time_calculator.py inside the A1 directory. In it, implement a function called time_calculator. This functions accepts a single parameter called seconds, and converts the seconds to days, hours, minutes, and seconds. You may assume that there are:

   (a) 60 seconds in a minute
   (b) 3,600 seconds in an hour
   (c) 86,400 seconds in a day.

   Print the number of days, hours, minutes and seconds **in that order** as integers separated by white space in a single line.

6. 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.

   Create a new file called compound_interest.py, and inside it implement a function called compound_interest. The function should accept the following parameters in this order:

   (a) principal (a float)
   (b) annual interest paid to the account (a float)
   (c) the number of times per year the interest is compounded (an int)
   (d) the number of years the account will be left alone to grow.

   Return A, the amount of money in the account after the elapsed time, as a float.

7. Add a file called random_game.py to the A1 folder. Inside this file, implement a function called rock_paper_scissors. This function accepts no input and returns no input. This function must allow the user to play one round of rock, paper, scissors with the computer. The function must generate a random number in the range [0, 2] to represent rock, paper, and scissors. Ask the user for their guess. If the user enters anything other than rock, paper, or scissors, print a helpful warning message. Otherwise, print out the computer's choice and tell the user if they won or not. Your function should clean the user input before trying to decide what they chose. That is, eliminate leading and trailing whitespace, normalize capitalization, etc.

8. Add a file called lotto.py to the A1 folder. Inside this file, implement a function called number_generator. This function must generate a lottery ticket containing a list of 6 unique numbers. Generate 6 unique random numbers in the range [1, 49] and print them in order from lowest to highest in a single line of output.

9. Add a file called phone_fun.py to the A1 folder. Inside this file, implement a function called number_translator. Many companies use phone numbers like 555-GET-FOOD, 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 ask the user to enter a 10-character telephone number in the format XXX-XXX-XXXX. Your function should return the telephone number with any alphabetical numbers translated into their numerical equivalent. For example, if the user enters 555-GET-FOOD, the function should return the string "555-438-3663".

## 6 Grading

Your first assignment will be marked out of 15. I will randomly select three (3) of the functions from this assignment and mark them. For full marks, you must:

1. (3) Correctly implement the functional requirements described in this document for each function I mark (2 marks per function).

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

3. (3) Describe the elements of computational thinking you applied when implementing each function I mark (1 mark per function).

4. (3) Correctly write and execute doctests for each function I mark that unambiguously describe the function's correct use, and correctly write and execute unit tests to thoroughly test each function I mark. Only unit test input that meets the precondition, or input that requires an error message.

5. (3) Correctly format and comment your code for each function I mark (1 mark per function). Eliminate all 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!), but sharing code or submitting someone else's work is not allowed.

Good luck, and have fun!