COMP 1510 Programming methods
Lab 10 and Assignment 4

Christopher Thompson
cthompson98@bcit.ca

BCIT CST — Due Sunday November 24th at 20:59:59 or earlier

Introduction

Welcome to your tenth COMP 1510 lab and the fourth COMP 1510 assignment.

Shh. Did you hear that? Was that sleigh bells? No. That was the sound of my nerves jangling from all the coffee I've been drinking. The Christmas season has come early to Vancouver judging by the shops, but for everyone at BCIT it is the end of term that draws nigh.

It's a busy time of year for students and faculty both. There are only three weeks of classes left, and we have presentations to present, projects to manage, labs to juggle, assignments to complete, and the marking oh the marking... But from just beyond the exams, the holidays taunt us with their promises of peace, presents, and conspicuous quantities of spirits, salubrity, and sleep.

In keeping with the "proto-holiday" spirit, early though it may be, I present you with the eight days of Python. Here be eight problems for the next eight days. These are important and fun idioms and algorithms I think you should know.

For this activity, you will:

1. Apply your detailed knowledge about functions, to wit, positional arguments, keyword arguments, default values, variable length parameter lists, and arbitrary keyword arguments
2. Write code that is well-documented, efficient, and parsimonious, i.e., minimizes complexity and length
3. Make decisions about where to put logic in order to minimize variable scope, minimize mutability, and make functions atomic
4. Write unit tests that demonstrate how your code units correctly respond to typical values in disjointed equivalency partitions and the boundaries between the partitions.

1 Submission requirements

1. **The lab portion must be complete before we start the quiz on Thursday this week.**
2. **The assignment portion must be completed by Sunday November 24th at 20:59:59 or earlier.**
3. Late deliverables will not be accepted for any reason.
4. 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.

2 Project Setup

Please complete the following:

1. For the lab portion, all of your code must go into the Lab10 folder of the the PyCharm project you created for the first lab.
2. After you complete each task, commit and push your change to version control. In order to earn full marks, you must commit and push after each function is complete. Your commit message must be a short message to me that describes what you completed for this commit, i.e., "deconstructed base_conversion", or "debugged function_name", etc.
3. Open the PyCharm project you created for the first assignment. Recall we made folders called A1, A2, ..., A5. Recall we have been using this project for ALL of our assignments this term.
4. For the assignment portion, all of your code must go into the A4 folder of the the PyCharm project you created for the first assignment.
5. When you are finished, you do not need to add me or your marker as collaborators, because you already invited us to collaborate with you.

3 Grading scheme

The lab portion of this activity will be marked out of 5:

1. 3 points for correctness of implementation
2. 1 point for efficiency of implementation. Full marks will be reserved for submissions that employ syntactic sugar like comprehensions to minimize code complexity and eliminate troublesome, cumbersome loops, etc.
3. 1 point for efficacy of testing

The assignment portion of this activity will be marked out of 10:

1. 1 point for code style
2. 6 points for correctness and efficiency of implementation (your actual code). Full marks will be reserved for submissions that employ syntactic sugar like comprehensions to minimize code complexity and eliminate troublesome, cumbersome loops, etc.
3. 3 points for your unit tests and (possibly) doc tests

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. For this assignment, **functions must contain no more than 15 lines of code** (excluding the definition statement and excluding any docstrings or doctests). 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 create functions that wrap around 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.
3. Each function must also be tested by a 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. 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: factorial of number

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

5 Implementation requirements

Implement the following:

1. There are eight problems in this document.
2. **You must solve one problem for the lab, and solve the other seven problems for the assignment.**
3. You may select any question you wish to solve in the lab.
4. A complete solution includes correct function(s), well-formatted docstrings, doctests where helpful, and unit tests.
5. The solution for question 1 must go in file question_1.py. The solution for question 2 must go in file question_2.py. And so on.
6. 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.

That's it! Good luck, and have fun!

5.1 On the first day of Python, my true love gave to me...

A sieve. A sieve of Eratosthenes. You must implement the sieve of Eratosthenes.

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 (your instructions are steps 8 through 10!):

1. Make a list of numbers: 0, 1, 2, 3, ..., N.
2. 0 is not prime, so cross it out.
3. 1 is not prime, so cross it out.
4. 2 is prime, but its multiples are not prime, so cross out 4, 6, 8, 10, ...
5. 3 is prime, but its multiples are not prime, so cross out 6, 9, 12, 15, ...
6. (4 is crossed out so next is 5) 5 is prime, but its multiples are not primes, so cross out 10, 15, 20, 25, ...
7. 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 list of prime numbers!
8. Implement a function called eratosthenes. This function will accept a single positive integer called upperbound. If the value passed to the eratosthenes function is not a positive integer, raise an appropriate exception.
9. Use the Sieve of Eratosthenes to determine which of the numbers in the range [0, upperbound] are prime numbers.
10. Return the primes between [0, upperbound] as a list. For example:

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

Figure 1: *The sieve algorithm in colour...*

5.2 On the second day of Python, my true love gave to me...

A greatest common divisor function. This is brought to you by Euclid, another ancient Greek polymath.

Write a function that finds the greatest common divisor (GCD) of two integers. A GCD for two numbers a and b is the largest number that cleanly divides into both a and b. For example, the greatest common divisor of 10 and -25 is 5. The greatest common divisor of 4 and 2 is 2. And the greatest common divisor of any two prime numbers, positive or negative, is always 1.

Euclid's method works like this (your instructions are steps 9 through 11!). Note also that there is some great pseudocode for Euclid's method online which you can implement with a while loop and a few variables:

1. gcd(a, 0) = a by definition.
2. gcd(a, b) = gcd(b, a mod b) where (a mod b) = a - b * floor(a / b).
3. For example, if we are finding the GCD of 270 and 192, we will start by dividing 270 by 192 which goes in once and generates a remainder of 78.
4. Next, we will divide 192 by 78 which equals 2, with a remainder of 36.
5. Divide 78 by 36, which divides 78 twice and generates a remainder of 6.
6. Divide 36 by 6, which divides 36 six times and leaves a remainder of 0.
7. gcd(6, 0) equals 6 by definition, so...
8. We can say that gcd(270, 192) = gcd(192, 78) = gcd(78, 36) = gcd(36, 6) = gcd(6, 0) = 6.
9. Create a function called gcd. This function will accept two non-zero integers called a and b. If either value passed to the gcd function is not a non-zero integer, raise an appropriate exception.
10. Use Euclid's method to determine the GCD of a and b.
11. Return the GCD. For example:

```
the_gcd = gcd(-25, 15)
print(the_gcd)
5
```

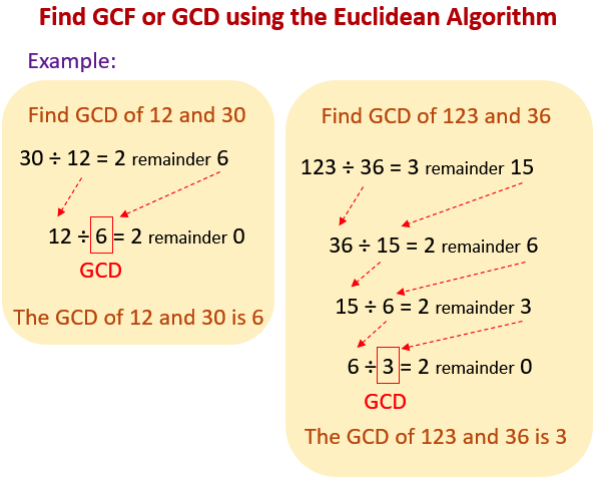

Figure 2: *A manual description of Euclid's algorithm*

5.3 On the third day of Python, my true love gave to me...

A fun brainteaser from Edsgar Dijkstra.

Edsgar Dijkstra, whom you met during our first history lessons so many weeks ago, developed a fun problem he called the Dutch National Flag problem. Given a list of strings, each of which is either 'red', 'white', or 'blue' (each is repeated several times in the list), rearrange the list so that the strings are in the order of the Dutch national flag: all the 'red' strings first, then all the 'white' strings, then all the 'blue' strings.

1. Write a function called dijkstra (watch your spelling here) that accepts a non-empty list that contains randomly shuffled strings 'red', 'white', and 'blue'.
2. Sort the strings in the original list. Do not make a copy and do not use any other data structures. Your code must be efficient.
3. There is no return value.

```
dutch = ['white', 'blue', 'blue', 'red', 'white', 'red', 'white']
dijkstra(dutch)
print(dutch)
['red', 'red', 'white', 'white', 'white', 'blue', 'blue']
```

5.4 On the fourth day of Python, my true love gave to me...

A simple sorting algorithm.

Sorting things in Python is easy. We're lucky that there is a built-in sort function that seems to work for most of our sequence collection needs. But there are different sorting algorithms that we should know about. Let's implement a simple search algorithm called selection sort:

1. Write a function called selection_sort (watch your spelling here) that accepts a non-empty list of sortable items and returns a sorted copy.
2. Raise an error if the argument passed to the function is not a non-empty list of sortable items.
3. The selection sort algorithm is not particularly efficient or fast, but it is very logical and easy to understand. Start by searching linearly through the list from beginning to end to find the smallest item.
4. Put that smallest item in index 0, and put whatever is in index 0 at the location where you found the smallest item. That is, swap them.
5. We now have a list which is divided into two conceptual sections: a sorted section of length 1 at the beginning, and 'the rest'.
6. Now starting at index 1, find the smallest item in the unsorted section of the list. Don't look at index 0, we already know that's the smallest.
7. Swap the (second-)smallest item we just found with the item in index 1.
8. Now starting at index 2, find the smallest item in the unsorted section of the list. Don't look at indices 0 and 1, we already know that the first two indices of the list are sorted correctly.
9. Keep doing this until we reach the end of the list. Your program should behave like this:

```
unsorted = [3, 5, 1, 9, -4]
sorted_copy = selection_sort(unsorted)
print(sorted_copy)
[-4, 1, 3, 5, 9]
```

5.5 On the fifth day of Python, my true love gave to me...

Money. Cold hard cash. To count.

1. Your task today is to implement a function called cash_money. The cash_money 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. For example, if 66.53 is entered, the program should return a dictionary that looks like this:

   ```
   breakdown = cash_money(66.53)
   print(breakdown)
   {50: 1, 10: 1, 5: 1, 1: 1, 0.25: 2, 0.01: 3}
   ```

2. If the value entered is not a positive double, raise an error.

3. We will consider the following denominations: 100, 50, 20, 10, 5, 2, 1, 0.25, 0.10, 0.05, 0.01 (I know the penny is gone, but let's pretend).

5.6 On the sixth day of Python, my true love gave to me...

Some more unit testing. Hooray!

Your task for today is to come up with tests for a hypothetical function called line_intersect, which accepts two lines as input and returns their intersection. Specifically:

1. Lines are represented as pairs of distinct points, such as [[0.0, 0.0], [1.0, 3.0]].

2. If the two lines don't intersect, line_intersect returns None.

3. If the two lines intersect in one point, line_intersect returns the point of intersection, such as [0.5, 0.75].

4. If the two lines are coincident (that is, lay on top of one another), the function returns its first argument (that is, a line).

5. What are the six most informative unit tests you can think of? That is, if you are ONLY allowed to write six unit tests. which would tell you the most about whether the function has been correctly implemented? (Consider all the possible values...)

6. Write your solution as a multi-line comment inside file question_06.py.

Figure 3: *Have you ever played pick up sticks?*

5.7 On the seventh day of Python, my true love gave to me...

Some refactoring. Bring it on!

1. Take a peek at this program. It's a script. It works, but it has no tests, and it contains no functions (and therefore no reusable code):

```
# Global Constant
_calories = {"lettuce": 5, "carrot": 52, "apple": 72, "bread": 66,
             "pasta": 221, "rice": 225, "milk": 122, "cheese": 115,
             "yogurt": 145, "beef": 240, "chicken": 140, "butter": 102
             }

# Input loop
new_item = input("Enter food item to add, or 'q' to exit: ")
while new_item != "q":

    new_item_calories = int(input("Enter calories for " + new_item + ": "))
    _calories[new_item] = new_item_calories

    total_calories = 0
    for item in _calories:
        total_calories = total_calories + _calories[item]

    food_item_names = []
    for item in _calories:
        food_item_names.append(item)

    avg_calories = total_calories / len(_calories)

    print("\nFood Items:", sorted(food_item_names))
    print("Total Calories:", total_calories,
          "Average Calories: %0.1f\n" % avg_calories)

    new_item = input("Enter food item to add, or 'q' to exit: ")
```

2. Copy the code to a new file in your project called question_07.py
3. Apply refactorings to convert the script to an elegant collection of short, precise, atomic functions.
4. Write a small suite of unit tests to prove your functions work.

5.8 On the eighth day of Python, my true love gave to me...

Imsomna. Thanks, Obama.

Do you have an alarm clock? Have you ever had insomnia and stared at it for hours? Have you ever noticed how the numbers are formed from segments (examine the figure below):

Figure 4: *Each number is composed of a few bars...*

1. Examine the sample image. The time 7:30 consistes of a 7, a 3, and a 0. The 7 contains 3 segments. The 3 contains 5 segments, and the 0 contains 6 segments. We can say that it requires 3 + 5 + 6 = 14 segments to represent 7:30 on an alarm clock. Turn the page.
2. Write a function called im_not_sleepy that determines what time(s) requires the highest number of bars.
3. Assume we are using a 12-hour clock, and only consider the bars used to create numbers.
4. Your solution must be efficient and short. Use Python's data structures and Pythonic idioms to make this short and sweet.
5. The function must return a string that contains the time that requires the highest number of bars, and the number of bars it requires to the console.