COMP 202

Fall 2021

Assignment 1

Due: Wednesday, October 6^{th}, 11:59 p.m.

Please read the entire PDF before starting. You must do this assignment individually.

Question 1:	15 points
Question 2:	85 points
	100 points total

It is very important that you follow the directions as closely as possible. The directions, while perhaps tedious, are designed to make it as easy as possible for the TAs to mark the assignments by letting them run your assignment, in some cases through automated tests. While these tests will never be used to determine your entire grade, they speed up the process significantly, allowing the TAs to provide better feedback and not waste time on administrative details. Plus, if the TA is in a good mood while grading, then that increases the chance of them giving out partial marks. :)

To get full marks, you must follow all directions below:

- Make sure that all file names and function names are **spelled exactly** as described in this document. Otherwise, a 50% penalty per question will be applied.
- Make sure that your code **runs without errors**. Code with errors will receive a very low mark.
- Write your name and student ID in a comment at the top of all `.py` files you hand in.
- Name your variables appropriately. The purpose of each variable should be obvious from the name.
- **Comment your code.** A comment every line is not needed, but there should be enough comments to fully understand your program.
- Avoid writing repetitive code, but rather call helper functions! You are welcome to add additional functions if you think this can increase the readability of your code.
- Lines of code should NOT require the TA to scroll horizontally to read the whole thing.
- Vertical spacing is also important when writing code. Separate each block of code (also within a function) with an empty line.
- **Up to 30% can be removed for bad indentation of your code, omission of comments, and/or poor coding style (as discussed in class).**

Hints & tips

- **Start early.** Programming projects always take more time than you estimate!
- Do not wait until the last minute to submit your code. **Submit early and often**—a good rule of thumb is to submit every time you finish writing and testing a function.
- Write your code **incrementally**. Don't try to write everything at once. That never works well. Start off with something small and make sure that it works, then add to it gradually, making sure that it works every step of the way.
- Read these instructions and make sure you understand them thoroughly before you start. Ask questions if anything is unclear!

- Seek help when you get stuck! Check our discussion board first to see if your question has already been asked and answered. Ask your question on the discussion board if it hasn't been asked already. Talk to your TA during office hours if you are having difficulties with programming. Go to my office hours if you need extra help with understanding a part of the course content.
  - At the same time, beware not to post anything that might give away any part of your solution—this would constitute plagiarism, and the consequences would be unpleasant for everyone involved. If you cannot think of a way to ask your question without giving away part of your solution, then please drop by our office hours.
- If you come to see us in office hours, please do not ask "Here is my program. What's wrong with it?" We expect you to at least make an effort to start to debug your own code, a skill which you are meant to learn as part of this course. And as you will discover for yourself, reading through someone else's code is a difficult process—we just don't have the time to read through and understand even a fraction of everyone's code in detail.
  - However, if you show us the work that you've done to narrow down the problem to a specific section of the code, why you think it doesn't work, and what you've tried to fix it, it will be much easier to provide you with the specific help you require and we will be happy to do so.

Revisions

Sept 20: Corrected examples for `find_maximal_subplots` function.

Sept 21: Corrected return value in example for `calculate_cost` function.

Sept 22: Adjusted phrasing for `get_num_subplots_for_budget` function (should read 'spacing between subplots', not 'spacing between plots'). Also, undid part of the change made on Sept. 20 (spacing in the example for `find_maximal_subplots` is indeed 1.67m, not 2.0m).

Part 1 (0 points): Warm-up

Do **NOT** *submit this part, as it will not be graded. However, doing these exercises might help you to do the second part of the assignment, which will be graded. If you have difficulties with the questions of Part 1, then we suggest that you consult the TAs during their office hours; they can help you and work with you through the warm-up questions. You are responsible for knowing all of the material in these questions.*

Warm-up Question 1 (0 points)

Practice with Number Bases:

We usually use base 10 in our daily lives, because we have ten fingers. When operating in base 10, numbers have a `ones` column, a `tens` column, a `hundreds` column, etc. These are all the powers of 10.

There is nothing special about 10 though. This can in fact be done with any number. In base 2, we have each column representing (from right to left) 1,2,4,8,16, etc. In base 3, it would be 1,3,9,27, etc.

Answer the following short questions about number representation and counting.

1. In base 10, what is the largest digit that you can put in each column? What about base 2? Base 3? Base n?
2. Represent the number thirteen in base 5.
3. Represent the number thirteen in base 2.
4. What is the number 11010010 in base 10?

Warm-up Question 2 (0 points)

Logic:

1. What does the following logical expression evaluate to?

```
(False or False) and (True and (not False))
```

2. Let a and b be boolean variables. Is it possible to set values for a and b to have the following expression evaluate as *False*?

```
b or (((not a) or (not a)) or (a or (not b)))
```

Warm-up Question 3 (0 points)

Expressions: Write a program `even_and_positive.py` that takes an integer as input from the user and displays on your screen whether it is true or false that such integer is even, positive, or both.

An example of what you could see in the shell when you run the program is:

```
>>> %Run even_and_positive.py
Please enter a number: -2
-2 is an even number: True
-2 is a positive number: False
-2 is a positive even number: False

>>> %Run even_and_positive.py
Please enter a number: 7
7 is an even number: False
7 is a positive number: True
7 is a positive even number: False
```

Warm-up Question 4 (0 points)

Conditional statements: Write a program `hello_bye.py` that takes an integer as input from the user. If the integer is equal to 1, then the program displays `Hello everyone!`, otherwise it displays `Bye bye!`.

An example of what you could see in the shell when you run the program is:

```
>>> %Run hello_bye.py
Choose a number: 0
Bye bye!

>>> %Run hello_bye.py
Choose a number: 1
Hello everyone!

>>> %Run hello_bye.py
Choose a number: 329
Bye bye!
```

Warm-up Question 5 (0 points)

Void Functions: Create a file called `greetings.py`, and in this file, define a function called `hello`. This function should take one input argument and display a string obtained by concatenating *Hello* with the input received. For instance, if you call `hello("world!")` in your program you should see the following displayed on your screen:

```
Hello world!
```

- Think about three different ways of writing this function.
- What is the return value of the function?

Warm-up Question 6 (0 points)

Void Functions: Create a file called `drawing_numbers.py`. In this file create a function called `display_two`. The function should not take any input argument and should display the following pattern:

```
  22
2   2
   2
 2
22222
```

Use strings composed out of the space character and the character '2'.

- Think about two different ways of writing this function.
- Try to write a similar function `display_twenty` which displays the pattern '20'

Warm-up Question 7 (0 points)

Fruitful Functions: Write a function that takes three integers `x`, `y`, and `z` as input. This function returns `True` if `z` is equal to 3 or if `z` is equal to the sum of `x` and `y`, and `False` otherwise.

Warm-up Question 8 (0 points)

Fruitful Functions: Write a function that takes two integers `x`, `y` and a string `op`. This function returns the sum of `x` and `y` if `op` is equal to `+`, the product of `x` and `y` if `op` is equal to `*`, and zero in all other cases.

Part 2

The questions in this part of the assignment will be graded.

The main learning objectives for this assignment are:

- Correctly create and use variables.
- Learn how to build expressions containing different type of operators.
- Get familiar with string concatenation.
- Correctly use `print` to display information.
- Understand the difference between inputs to a function and inputs to a program (received from the user through the function `input`).
- Correctly use and manipulate inputs received by the program from the function `input`.
- Correctly use simple conditional statements.
- Correctly define and use simple functions.
- Solidify your understanding of how executing instructions from the shell differs from running a program.

Note that this assignment is designed for you to be practicing what you have learned in the videos up to and including Lecture 11 (Functions 3). For this reason, you are NOT allowed to use anything seen after Lecture 11 or not seen in class at all. You will be heavily penalized if you do so.

For full marks, in addition to the points listed on page 1, make sure to add the appropriate documentation string (docstring) to *all* the functions you write. The docstring must contain the following:

- The type contract of the function.
- A description of what the function is expected to do.
- At least 3 examples of calls to the function. You are allowed to use *at most one* example per function from this PDF.

Examples

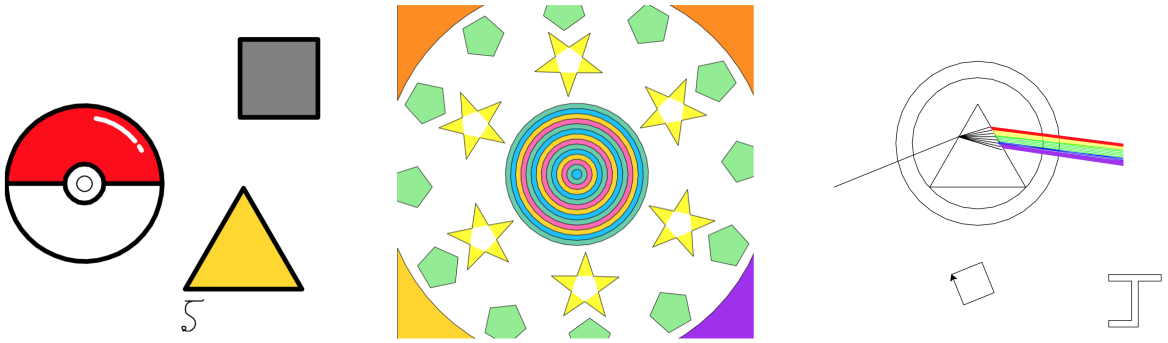
For each question, we provide several **examples** of how your code should behave. All examples are given as if you were to call the functions from the shell.

When you upload your code to codePost, some of these examples will be run automatically to test that your code outputs the same as given in the example. However, **it is your responsibility to make sure your code/functions work for any inputs, not just the ones shown in the examples.** When the time comes to grade your assignment, we will run additional, private tests that will use inputs not seen in the examples. You should make sure that your functions work for all the different possible scenarios. Also, when testing your code, know that mindlessly plugging in various different inputs is not enough—it's not the quantity of tests that matters, it's having tests that cover all of the possible scenarios, and that requires thinking about possible scenarios.

Furthermore, please note that your code files **should not contain any function calls in the main body of the program** (i.e., outside of any functions). Code that does not conform to this rule **will automatically fail the tests on codePost and thus be heavily penalized**. It is OK to place function calls in the main body of your code for testing purposes, but if you do so, make certain that you remove them before submitting. You can also test your functions by calling them from the shell.

Safe Assumptions

For all questions in this assignment, you can safely assume that the **type** of the inputs (both to the functions and those provided to the program by the user) will always be correct. For example, if a function takes as input a string, you can assume that a string will always be provided. At times you will be required to do some input validation, but this requirement will always be clearly stated. Otherwise, your functions should work with any possible input that respect the function's description. For example, if the description says that the function takes as input a positive integer, then it should work with all integers greater than 0. If it mentions an integer, then it should work for *any* integer. Make sure to test your functions for edge cases!



Question 1: Turtle Art (15 points)

This question asks you to write a function `my_artwork()` in a file called `artwork.py`. The function should take **no arguments** and draw a picture using the Turtle module. You are free to draw what you like, but your code/drawing must satisfy at least the following requirements:

- the drawing must include at least three shapes
- at least one shape must be drawn using a for loop
- at least one shape must be drawn using a function, with the function having at least two parameters that modify the shape being drawn in some way. (You cannot simply copy one of the functions we have written in our lectures - you must write your own function that is dissimilar to the ones seen in class.)
- the drawing must include at least two different colors
- everything must fit into the Turtle window (do not draw outside of the window or make the window larger)
- the first letter of your first name must appear somewhere (you must sign your artwork!)
- there should be **no** calls to the `input` function
- **As always, do not call any functions (including `turtle.Turtle`) in the main body.**

Any submission meeting these requirements will obtain full marks, but you are encouraged to go beyond them. Our TAs will be showcasing their favorite submissions in the Slack. (Due to McGill policy, we cannot share students' names without permission, so the chosen artworks will be posted in the Slack without names. However, if you would like your name to appear with your artwork if the TA decides to post it in the Slack, then please write a sentence to that effect in your `README.txt` file.)

Note: Recall that you can import the `speed` function from turtle and then call `speed("fastest")` to speed up the drawing routines, so that you don't waste time when testing your code.

Also, a reminder to please only use the functions from the Turtle module that we have seen in class, or you will lose marks. There is one exception: you can use the `circle` function from Turtle module. `circle(r)` takes a radius `r` as argument and draws a circle of the given radius. You can also specify a second integer argument for the extent of the circle to draw, e.g., `circle(r, 90)`, which will draw only a quarter of a circle (90 degrees).

Some submissions from students of previous years (with their size scaled down to fit) can be found at the top of this page.

Question 2: Sculpture Garden (85 points)

The Redpath Museum is planning a sculpture garden on the grounds of a large field on the West Island. (This is not actually true, but let us go along with it for the purposes of this question.)

There will be a certain number of sculptures arranged in a grid formation. Each sculpture will be enclosed by a fence.

The Museum is currently deciding how many sculptures to display in the garden, and how large a plot of land should be allocated for the garden, given that there should be a certain amount of space between each sculpture.

In this question, we will write some code to help the Museum with these decisions. For the purposes of this assignment, we will assume that the plot of land can have any size. Let us also assume that the Museum has many sub-basement levels where an infinite number of sculptures are stored, so neither plot size nor number of sculptures are bounded.

We will use the terms 'plot' and 'subplot' below. Plot refers to the total area of land for the sculpture garden. The plot will be rectangular and have a width and height (in metres). A subplot is a fence-enclosed rectangular area for a specific sculpture. Each subplot will have the same width and height. A plot can contain many subplots, depending on the size of the plot and the size of a subplot.

Our code will perform two tasks. One is to calculate the cost of the sculpture garden, given the size of a plot of land, the size of each sculpture's subplot, the spacing between each sculpture's subplot, and the fencing and gravel needed for each subplot. The other task will be to calculate how many sculptures could fit into a garden of a certain size, given a maximum budget.

These two tasks require a lot of different computations. Whenever you are faced with a problem like this, which will most often be the case when programming, it is best to break it down into smaller sub-parts, or functions. Put each different computation into its own function. This way of organizing serves two purposes. First, it makes your code file much easier to read through and navigate. Secondly, and most importantly, it lets you focus on one thing at a time. Each time you write a function, you should always **test it thoroughly**, giving it different inputs and checking that the output is what you expect. It is easier to do this when the functions are small and do not do too much on their own. Only once you are certain that it is working properly, then you can move onto the next function.

Functions also help in code re-use. By putting some computation into a function, if we then needed to perform the same computation again, we can just call that same function instead of copy-and-pasting the code to a different location. Copy-and-pasting is bad in programming! One of the keys of programming is **don't repeat yourself** ('DRY'). Whenever you have repeated several lines of code, ask yourself if there is a way to simplify the code by putting it into a function instead (and then just calling it when needed).

With the above in mind, we will begin writing our functions. Write all your code for this question in a file called `garden.py`.

First, define the following global variables at the top of your code:

- `BORDER_SPACING = 1.0` (represents the amount of spacing on the edges of the plot)
- `FENCE_COST_PER_METRE = 10.0` (the cost of fencing per metre)
- `GRAVEL_COST_PER_METRE_SQUARED = 2.0` (the cost of gravel per squared metre)
- `COLOR1_COST = 5.0` (the cost in dollars of painting the fence a certain color per metre)
- `COLOR2_COST = 5.0`
- `COLOR3_COST = 5.0`
- `COLOR4_COST = 10.0`
- `COLOR5_COST = 1000.0`

Note that the names of these global variables are in all-caps. Variables in all-caps are known by convention to be 'constant' variables, or 'constants.' A constant is a variable whose value will never change throughout the execution of the program (e.g., it does not depend on user input or on the result of any calculation). It is set once, typically at the top of your code file, and not modified afterwards in the code. (Note that it is always possible for you to modify the value of a constant yourself, by replacing the initial value on the line where it is defined.)

The examples shown in the functions below will assume that the constants are set to the above values. However, note that during testing, we may modify the values of these global variables and check that your code still works properly. You should make sure that your code uses these global variables and that you do not 'hard-code' the values (i.e., by writing the values above directly into the code instead of using the variables).

- `get_perimeter_of_subplot(subplot_w, subplot_h)`: Takes two floats as input (the width and height of a subplot) and returns the perimeter of the subplot (as a float).

```
>>> get_perimeter_of_subplot(5, 5)
20

>>> get_perimeter_of_subplot(1, 10)
22
```

- `get_area_of_subplot(subplot_w, subplot_h)`: Takes two floats as input (the width and height of a subplot) and returns the area of the subplot (as a float).

```
>>> get_area_of_subplot(5, 5)
25

>>> get_area_of_subplot(1, 10)
10
```

- `get_cost_per_subplot(subplot_w, subplot_h, color_cost_per_metre)`: Takes three floats as input (the width and height of a subplot, and the cost per metre of a fence color) and returns the cost of that subplot (as a float). There are two parts to the cost: the cost of the fencing and the cost of the gravel. To calculate the cost of fencing, take the product of the perimeter of the subplot with the cost per metre of fencing and cost per metre of the fence color. To calculate the cost of the gravel, take the product of the area of the subplot with the cost per metre squared of gravel. There is no cost for the sculpture itself as they are already owned by the Museum.

```
>>> get_cost_per_subplot(5, 5, 5)
350.0

>>> get_cost_per_subplot(1, 10, 1)
262.0
```

- `get_cost_for_color(color_choice)`: Takes a positive integer as input, representing a color, and returns a float corresponding to the cost per metre of painting a fence that color. If the integer is 1, then the value in the corresponding global variable (`COLOR1_COST`) should be returned, and similarly for integers 2 through 5. If the integer is not between 1 and 5, then 0.0 should be returned.

```
>>> get_cost_for_color(1)
5.0

>>> get_cost_for_color(9001)
0.0
```

- `choose_color()`: This function takes no inputs. It prints the list of five available colors to the user. The user will then select one color by typing a number from 1 to 5, and the function will return the cost of the given color (as a float). You are free to choose the names of the five colours yourself (but you must use the prices in the global variables as defined above).

  (Note: Text shown below with a light-gray background corresponds to input entered by the user.)

```
>>> num = choose_color()
Color options
1        Martlet Red
2        Montreal Orange
3        Cosmic Yellow
4        The Fourth Colour
5        Vantablack
What color would you like? 5

>>> print(num)
1000.0
```

- `get_num_subplots(plot_w, plot_h, subplot_w, subplot_h, spacing_w, spacing_h)`: This function will calculate the maximum possible number of subplots (rectangular fenced areas for sculptures) given the inputs. The inputs to the function are the width and height of the plot, the width and height of each subplot (sculpture area), and the width and height that should separate each subplot (the spacing). All inputs are floats and are in metres. The function will return an integer corresponding to the maximum number of subplots that can fit. Consider the following for this function:
  - The subplots should be arranged in grid formation, i.e., in rows and columns. The number of columns can be calculated by taking the width of the plot, subtracting the border spacing on either side, and then dividing by the width of a subplot and associated spacing. That is, if the plot width is 10m and border spacing is 1m on either side, then there is an effective width of 8m. If each subplot is 1m wide, with 1m spacing, then 4 subplots can fit side-by-side. Then the same operation can be done for the rows (by using height instead of width).
  - However, there is one corner case. Consider the above example, but with a plot width of 11m, or 9m after the 1m border spacing on either side. Then, 4 subplots could fit, but there would be 1m left over. A subplot of 1m and spacing of 1m could not fit here, but seeing as how there is already a border spacing at the end anyway, then we could in fact fit in the subplot, without spacing, while still respecting the spacing requirements. Therefore, in this case 5 subplots can fit. Make sure to take this into account for the calculation of the number of subplots for rows as well as for columns.

```
>>> get_num_subplots(10, 3, 1, 1, 1, 1)
4

>>> get_num_subplots(11, 3, 1, 1, 1, 1)
5
```

- `calculate_cost()`: This function takes no inputs. It will ask the user to enter the following values: the width and height of the plot, width and height of a subplot, width and height of spacing between the plots, and the color that the fences should be painted (the latter as an integer between 1 and 5). It will then calculate the number of subplots that can fit in the plot, as well as the total cost for the fencing. It will print out these two values, with the cost rounded to two decimal places, and will then return the cost (as a float), not rounded.

  Note: Even if there is zero spacing between subplots, each subplot will still have its own fence on all four sides, including the shared sides. That is, there are no shared fences.

```
>>> cost = calculate_cost()
Enter the width of the plot: 10
Enter the height of the plot: 10
Enter the width of a subplot: 1
Enter the height of a subplot: 1
Enter the horizontal spacing between subplots: 1
Enter the vertical spacing between subplots: 1
Color options
1        Martlet Red
2        Montreal Orange
3        Cosmic Yellow
4        The Fourth Colour
5        Vantablack
Which color would you like? 5
16 subplots can fit in the plot, with a total cost of $64672.0

>>> print(cost)
64672.0
```

We will now write functions for our second task: to calculate how many sculptures could fit into a garden of a certain size, given a maximum budget. Place them in the same file as the above.

Note: For the two functions below, we will assume that the plots and subplots are squares, that is, the width of a plot is equal to its height, and the same for a subplot. We will call this dimension the length.

- `get_num_subplots_for_budget(plot_length, subplot_length, budget, color_cost_per_metre)`: This function takes four inputs, all floats: the length of the plot, length of a subplot, total budget, and the cost per metre of a fence color. It calculates how many subplots can fit into the given plot size, costing at most the given budget. (Do not consider spacing between subplots for this function.) It then returns the resulting number as an integer.

  Note: To calculate the number of subplots, you can divide the budget by the cost of a subplot. However, make sure that the resulting number of subplots does not exceed the available space inside the plot. If it does, then you must decrease the number just enough so that the plot is filled maximally.

```
>>> get_num_subplots_for_budget(3, 1, 46, 1)
1

>>> get_num_subplots_for_budget(10, 1, 250, 5)
4
```

- `find_maximal_subplots()`: This function takes no inputs. It will ask the user to enter the following values: the length of a plot and of a subplot, the color that the fences should be painted, and the total budget. It will then calculate the number of subplots that can fit in the plot, costing at most the given budget, and print the integer to the screen. Given that number of subplots, it will then calculate how much spacing can be between them, and then print that number (as a float rounded to two decimal places) to the screen. Recall that a plot will have a fixed amount of spacing on the border; this spacing should not be included in the aforementioned. The number of subplots calculated should then be returned (as an integer).

```
>>> num_subplots = find_maximal_subplots()
Enter the side length of the plot: 10
Enter the side length of a subplot: 1
Color options
1        Martlet Red
2        Montreal Orange
3        Cosmic Yellow
4        The Fourth Colour
5        Vantablack
Which color would you like? 1
Enter your maximum budget: 5000
With the given budget, 64 subplots can fit in the plot, with spacing of 0.0m.

>>> print(num_subplots)
64

>>> num_subplots = find_maximal_subplots()
Enter the side length of the plot: 10
Enter the side length of a subplot: 1
Color options
1        Martlet Red
2        Montreal Orange
3        Cosmic Yellow
4        The Fourth Colour
5        Vantablack
Which color would you like? 1
Enter your maximum budget: 200
With the given budget, 3 subplots can fit in the plot, with spacing of 1.67m.

>>> print(num_subplots)
3
```

Finally, we will write a function that prints the available options to the user and asks them to select one to perform.

- `menu()`: This function takes no inputs and does not return anything. It prints the two options available to the user and asks them to enter a number corresponding to their desired option. It then executes the appropriate option. If the user enters an invalid number, `Invalid choice.` should be printed and the program should end. When the program ends, `Have a nice day!` should be printed to the user.

```
>>> menu()
Welcome to the Plot Calculator!
Please choose from the following:
1        Calculate cost of plot project
2        Find maximal subplots
Your choice: 1
Enter the width of the plot: 10
Enter the height of the plot: 10
Enter the width of a subplot: 1
Enter the height of a subplot: 1
Enter the horizontal spacing between subplots: 1
Enter the vertical spacing between subplots: 1
Color options
1        Martlet Red
2        Montreal Orange
3        Cosmic Yellow
4        The Fourth Colour
5        Vantablack
Which color would you like? 5
16 subplots can fit in the plot, with a total cost of $64672.0
Have a nice day!

>>> menu()
Welcome to the Plot Calculator!
Please choose from the following:
1        Calculate cost of plot project
2        Find maximal subplots
Your choice: 3
Invalid choice.
Have a nice day!
```

What To Submit

You must submit all your files on codePost (https://codepost.io/). The files you should submit are listed below. Any deviation from these requirements may lead to lost marks.

`artwork.py`
`garden.py`
`README.txt` In this file, you can tell the TA about any issues you ran into while doing this assignment. If you point out an error that you know occurs in your program, it may lead the TA to give you more partial credit.
Remember that this assignment like all others is an `individual` assignment and must represent the entirety of your own work. You are permitted to verbally discuss it with your peers, as long as no written notes are taken. If you do discuss it with anyone, please make note of those people in this `README.txt` file. If you didn't talk to anybody nor have anything you want to tell the TA, just say "nothing to report" in the file.

You may make as many submissions as you like, but we will only grade your final submission (all prior ones are automatically deleted).

Note: If you are having trouble, make sure the names of your files are exactly as written above.