CSC110 Fall 2022 Assignment 3: Loops, Mutation, and Applications

Print this handout

In this assignment, you'll take what you learned about using data classes, loops, and mutation over the past few weeks, and apply these programming techniques to a few new problem domains. In Part 1, you'll perform a data analysis on some CSV files containing Toronto health and low income status data In Part 2, you will learn about *sentiment analysis* and debug a small program that analyses the sentiments found in movie reviews. And finally, in Part 3 you will use loops to create some interactive visualizations of 2-D point sequences that result in some truly beautiful patterns. What an exciting assignment!

Logistics

- Due date: Wednesday, November 2 before 12pm noon Eastern Time.
  - We are using **grace credits** to allow you to submit your work up to 30 hours after the due date. Please review the Policies and Guidelines: Assignments page (https://q.utoronto.ca/courses/278340/pages/policies-and-guidelines-assignments?module_item_id=3928858) carefully!
- This assignment must be done **individually**.
- You will submit your assignment solutions on MarkUs (see Submission instructions at the end of this handout).

Starter files

To obtain the starter files for this assignment:

1. Login to MarkUs (https://markus.teach.cs.toronto.edu/2022-09) and go to CSC110.
2. Under the list of assignments, click on **Assignment 3**.

3. On the assignment page, click on the **Download Starter Files** button (near the bottom of the page). This will download a zip file to your computer.
4. Extract the contents of this zip file into your `csc110/assignments/` folder.
5. You should see a new `a3` folder that contains the assignment's starter files!

You can then see these files in your PyCharm `csc110` project, and also upload the `a3.tex` file to Overleaf to complete your written work.

General instructions

This assignment contains a mixture of both written and programming questions. All of your written work should be completed in the `a3.tex` starter file using the LaTeX typesetting language. You went through the process of getting started with LaTeX in the Software Installation Guide (https://q.utoronto.ca/courses/160038/pages/setting-up-your-computer-start-here?module_item_id=1346385), but for a quick start we recommend using the online platform Overleaf (https://www.overleaf.com/) for LaTeX. Overleaf also provides many tutorials (https://www.overleaf.com/learn/latex/Learn_LaTeX_in_30_minutes) to help you get started with LaTeX.

> **Tip**: See this video tutorial (https://www.overleaf.com/learn/how-to/How_to_upload_files_to_a_new_project) for instructions on how to upload a `.tex` file to Overleaf.

Your programming work should be completed in the different starter files provided (each part has its own starter file). We have provided code at the bottom of each file for running doctest examples and PythonTA on each file. We are **not** grading doctests on this assignment, but encourage you to add som as a way to understand and test each function we've asked you to complete. We *are* using PythonTA to grade your work, so please run that on every Python file you submit using the code we've provided.

> **Warning**: one of the purposes of this assignment is to evaluate your understanding and mastery of the concepts that we have covered so far. So on this assignment, you may only use parts of the Python programming language that we have covered in Chapters 1-6 of the Course Notes. Other parts are not allowed, and parts of your submissions that use them may receive a grade as low as **zero** for doing so.

Python package installation: `pandas`

Completing Part 1 of this assignment requires one additional Python package, `pandas` (the lastest versions, 1.5.0 or 1.5.1). You have two options for installing it (both in PyCharm):

1. Open the `requirements.txt` file we provided in the `csc110` folder and add `pandas` on a new line. Then, you should see a prompt to "install missing packages" (you may need to restart PyCharm to have it detect the change).
2. Alternatively, open the Settings/Preferences window and then:
    a. Go to Project: csc110 -> Python interpreter
    b. Click on the "+" icon and search for `pandas` and install it.

Part 1: Data Analysis with Toronto Health

A commonly-held belief is that an individual's health is largely influenced by the choices they make. However, there is lots of evidence that health is affected by systemic factors.

Health researchers often study the relationships between an individual's health outcomes and factors related to their physical environment, social and economic situations, and geographic location. Studies such as this one investigate how a particular health outcome (living with hypertension) are tied to a systemic factor (the income level of a country) (https://doi.org/10.1161/CIRCRESAHA.120.318729).

In this part, you will write code to assist with analyzing data on the relationship between **hypertensio** (also known as high blood pressure) and **income levels** in Toronto neighbourhoods. The datasets you will work with represent real data obtained from the Ontario Community Health Profiles Partnership (https://www.ontariohealthprofiles.ca/dataTablesON.php?varTab=HPDtbl&select1=7), though we hav simplified the files for this assignment.

Part 1(a): Reading and modeling the data

To start, please read the following descriptions of the two types of datasets for this part of the assignment.

> *Note*: all dataset files described below can be found in your `a3/datasets/part1/` folder.

Neighbourhood hypertension datasets

You are given two comma-separated values (CSV) files for the City of Toronto for the hypertension rates in each of Toronto's neighbourhoods as of April 1, 2017. The full dataset is found in `hypertension_data_2016.csv`, while a small subset of the data is in `hypertension_data_small.csv`; we've provided the latter to help you test your work.

The first row in the CSV file contains header information, and the remaining rows each contain data for a single Toronto neighbourhood relating to its population and hypertension prevalence by age group.

Column information for neighbourhood hypertension datasets.

Column index	Description
0	The name of the neighbourhood. Neighbourhood names are unique.
1	The number of people aged 20 and older with hypertension in the neighbourhood.
2	The total number of people aged 20 and older in the neighbourhood.
3	The number of people aged 20 to 44 with hypertension in the neighbourhood.
4	The total number of people aged 20 to 44 in the neighbourhood.
5	The number of people aged 45 to 64 with hypertension in the neighbourhood.
6	The total number of people aged 45 to 64 in the neighbourhood.
7	The number of people aged 65 and older with hypertension in the neighbourhood.
8	The total number of people aged 65 and older in the neighbourhood.

Neighbourhood low income data files

You are given two comma-separated values (CSV) files for the City of Toronto for the low income population rates in each of Toronto's neighbourhoods based on the Canadian 2016 Census. The full dataset is found in `low_income_data_2016.csv`, while a small subset of the data is in `low_income_data_2016_small.csv`; we've provided the latter to help you test your work.

The first row in the CSV file contains header information, and the remaining rows each contain data for a single Toronto neighbourhood relating to its population and low income population rates.

Here is a description of the different columns of the dataset:

Column information for neighbourhood low income population rates.

Column index	Description
0	The name of the neighbourhood. Neighbourhood names are unique.
1	The total population in the neighbourhood.
2	The number of people in the neighbourhood with low income status.

Dataset notes

1. The names of the Toronto neighbourhoods are standardized, and so you may assume they are the same between the hypertension and low income datasets.
2. You should *not* assume that the neighbourhoods appear in the same order across the two types of datasets (and you should not modify the CSV files to sort them yourself!).
3. The neighbourhood population numbers in the low income datasets includes people younger than 20, while the hypertension datasets do not. This means you shouldn't expect the population totals to match across the two datasets. (And in fact, two datasets counted populations at different times, so we'd expect them to be different even if people younger than 20 were included in the hypertension datasets.)

After you've read the dataset descriptions, open `a3_part1.py` and complete the following tasks.

1. Read through the two data classes `HypertensionData` and `LowIncomeData`, which we have provided for you. You will use these data classes to represent a single row (i.e., one neighbourhoo of the hypertension and low income datasets, respectively.
2. Implement the functions `read_hypertension_data` and `read_low_income_data`, which are responsible for reading data from the datasets we've provided you.

Part 1(b): Computations

To better understand these datasets and practice your learning with for loops and mutation, we have provided some functions for you to complete under the "Part 1(b)" heading.

For each of these functions:

1. Complete the provided doctest.

2. Implement the function body.

You **must** use at least one for loop with an appropriate accumulator to implement each function, and you **may not** use comprehensions.

Part 1(c): Comparing hypertension and low income rates

Now, let's return to the question at hand: comparing the neighbourhood hypertension and low income rates. To do so, we need to combine information from each of the two datasets.

1. First, review the provided `CombinedRateData` data class definition, and then implement the function `combine_rates`.

   You are not required to add doctests for this function, though you may do so. We strongly recommend testing this function, *at the very least in the Python console*, before moving on.

   As in Part 1(b), you **must** use a for loop to implement this function, and **may not** use comprehensions.

2. Read through the provided code in `plot_combined_rates`. This code uses a new function calle `scatter` imported from `plotly.express`. plotly (https://plotly.com/python/) is a powerful Python library for visualizing data, and we'll be using it a few times this year.

   You should not modify this function, but you are responsible for reading through it to understar the different arguments to `plotly.express.scatter`.

3. Finally, putting everything together! Complete the `part1_example` function, which takes in file paths for hypertension and low income neighbourhood datasets, and generates a scatterplot visualization of the hypertension and low income rates in each neighbourhood.

   Here are two examples of how you can call this function in the Python console:

   ```
   >>> part1_example('datasets/part1/hypertension_data_small.csv',
           'datasets/part1/low_income_data_small.csv', '20+')
   >>> part1_example('datasets/part1/hypertension_data_small.csv',
           'datasets/part1/low_income_data_small.csv', '65+')
   ```

   When you're confident about those two calls, you can try replacing the small datasets with the complete Toronto neighbourhood datasets.

Now, if you're just looking at the scatterplots produced by plotly, you might not feel very satisfied how can we tell whether there's a relationships between low income and hypertension rates in a given neighbourhood? We've reached the end of the required tasks for Part 1 (which are focused c computation), but if you're interested in digging a bit deeper into the *data science*, please read th text below.

(OPTIONAL) Trendlines and standardization

One useful feature of plotly's scatterplots is the ability to add *trendlines* to a graph. To do so:

1. Install the `statsmodels` Python package.
   - In PyCharm, go to Settings/Preferences -> Project: csc110 -> Python interpreter.
   - Click on the "+" icon and search for `statsmodels` and install it.
2. Then, modify `plot_combined_rates` by adding an additional argument to `scatter:`

```
figure = scatter(
    ...,  # existing code, but ADD a comma at the end
    trendline='ols'
)
```

   Then when you call this function, you'll see both a scatterplot of the data and a *line of best fit* which has been computed using linear regression, which we covered in [4.8 Application: Linear Regression (https://www.teach.cs.toronto.edu/~csc110y/fall/notes/04-function-specification-and-correctness/08-linear-regression.html)](https://www.teach.cs.toronto.edu/~csc110y/fall/notes/04-function-specification-and-correctness/08-linear-regression.html).

   Note: you can keep or remove the `trendline='ols'` argument when you submit this file, either way will be accepted.

Now, some analysis!

3. Try calling the `part1_example` on the Toronto neighbourhood datasets with an `age_group` of `'20+'`. You should see something unexpected: there seems to be a slight *negative* correlation between low income and hypertension rates.

Intuitively, there are many other differences between the neighbourhoods beyond their "low income rate", so perhaps this isn't too surprising. One big difference that we can see is that different neighbourhoods have different proportions of people in each age group, and older

people generally have higher rates of hypertension than younger people.

4. Try calling `part1_example` with `'65+'` instead, so that we only compare the hypertension rates for this age group across neighbourhoods. You should see a dramatic difference: now there is indeed a *positive* correlation between low income rate and hypertension rate.

To remove the effect of differing age group sizes in health analysis, one common technique is called **age standardization**, which computes an adjusted rate by scaling age groups so that they are the same size across all different populations (e.g., different Toronto neighbourhoods). You can read more about age standardization at https://www.statcan.gc.ca/en/dai/btd/asr (https://www.statcan.gc.ca/en/dai/btd/asr).

Part 2: Loops and Mutation Debugging Exercise

A movie review typically includes a critic's comments and a score, but the score doesn't portray the sentiment and raw emotion in the critic's comments. Building on their success as restaurant owners (../../a1/handout/), David and Tom have decided to build their own *movie review platform* that labels reviews as positive, negative, or neutral based on the intensity of the words used in the review.

They use the VADER Lexicon (https://github.com/cjhutto/vaderSentiment), which we represent as a mapping from words to a positive/negative intensity score. For example, here are two words and their intensities:

Word	Intensity
awesome	3.1
awful	-2.0

The *polarity* of a review is one of `{'positive', 'negative', 'neutral'}`, and is calculated by finding the average intensity of the lexicon words used in the review. Words that don't appear in the VADER Lexicon are ignored when calculating the average.

Polarity	Average intensity
`positive`	≥ 0.05
`neutral`	-0.05 to 0.05, exclusive

Polarity	Average intensity
`negative`	≤ -0.05

In `a3_part2.py`, David and Tom have written a small program to read review data from a csv file, extract the lexicon words from the review, and then compute review's the average intensity and polarit They have painstakingly gone through three reviews and manually calculated their polarity and averag intensity. Unfortunately, when they compare the calculated values to the values computed by the program, David and Tom find that their program has some errors!

Answer the following questions about this program and `pytest` report. Write your responses in `a3.tex`, except for 2(b), which you will complete in `a3_part2.py` directly.

1. Run `a3_part2.py` to generate a `pytest` report. Note that we've already provided the code for running `pytest` in the main block at the bottom of the file.

   Based on the report, state which tests, if any, passed, and which tests failed. Just state the name o the tests and whether they passed/failed, and do not give explanations.

2. For *each* failing test from the `pytest` report:

   a. Explain what error is causing the test to fail.

      Hint: each test refers to a data file under `datasets/reviews`. The last column of each csv file stores the test of the review.

   b. Edit `a3_part2.py` by fixing the function code so that all tests pass. Unlike Assignment 1, th tests themselves do *not* contain errors.

      The changes should be small and must be to fix errors only; the original purpose of all functions and tests must remain the same. The expected value in the tests is correct, do not change it.

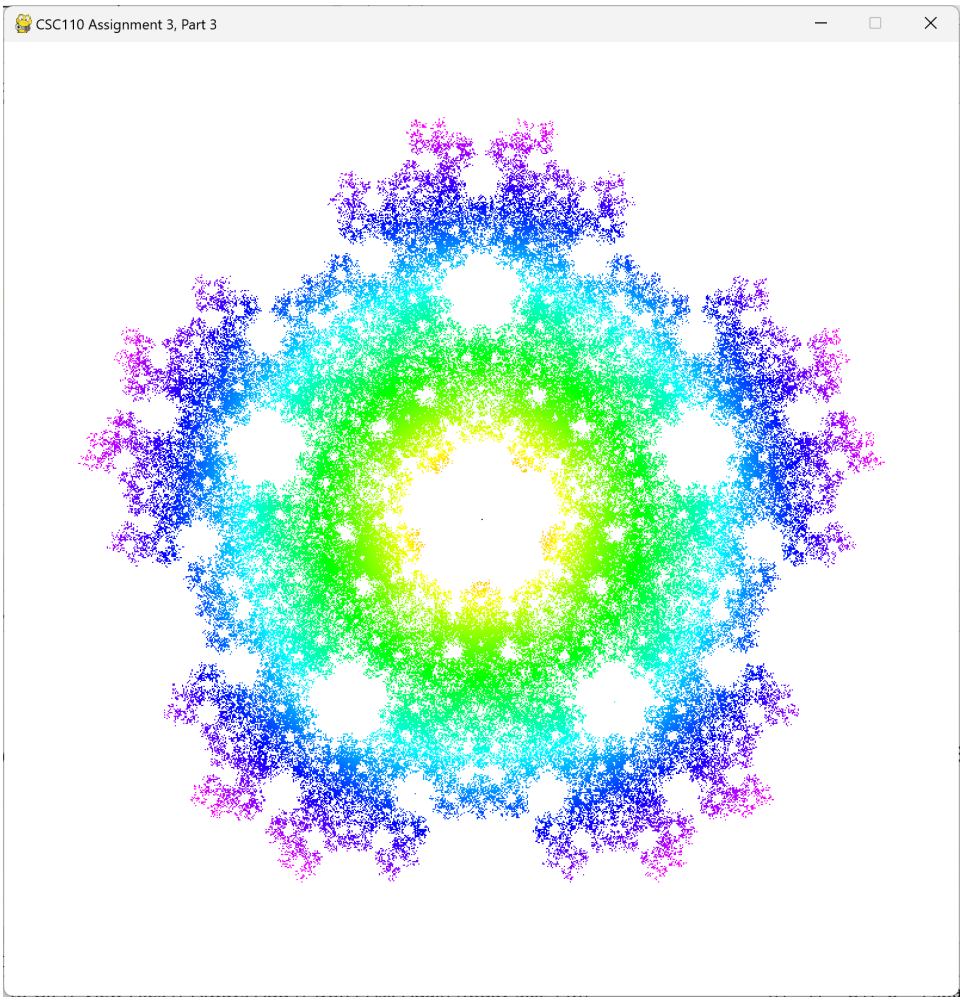
3. For *each* test that passed on the original code (before your changes in Question 2), explain why that test passed even though there were errors in the Python file.

Part 3: Chaos, Fractals, Point Sequences (or, drawing beautiful patterns with loops!)

In this final part of Assignment 3, you'll explore a fascinating branch of mathematics known as *chaos theory*. From the Wikipedia page (https://en.wikipedia.org/wiki/Chaos_theory):

> Chaos theory is an interdisciplinary scientific theory and branch of mathematics focused on underlying patterns and deterministic laws, of dynamical systems, that are highly sensitive to initial conditions, that were once thought to have completely random states of disorder and irregularities. Chaos theory states that within the apparent randomness of chaotic complex systems, there are underlying patterns, interconnection, constant feedback loops, repetition, self-similarity, fractals, and self-organization.

To make that more concrete, you'll use loops to generate a very simple kind of "dynamical system": a sequence of (x, y) points on the Cartesian plane. You'll see that from even very simple rules that govern how points are generated, some truly beautiful patterns can emerge.



After seeing that image of a Pygame window, We hope you're excited to get started! Begin by opening the `a3_part3.py` starter file; you'll complete all your work for this part in that file. You should also keep `a3_helpers.py` open while working on this part.

1. Your first task is to complete the `part3_warmup` function by following the instructions in its docstring.

   Note: when you run this function, an 800-by-800 Pygame window should appear. Depending on your screen resolution, the pixels may be very small, so you may need to look carefully to see them

Now, we'll introduce the concept of a *point sequence*, which is a type of "dynamical system" studied by chaos theory.

Point Sequence 1

1. To start, we choose a list of at least three **vertex points** (also called **vertices**). For this example, we'll use the vertices of a square, starting at the top-left corner (in Pygame coordinates) and going clockwise:

   $$VERTICES = [(100, 100), (700, 100), (700, 700), (100, 700)]$$

2. Next, we choose a starting point p_0. For our example, we'll choose $p_0 = (500, 500)$, though we could have chosen any point. Note that p_0 does *not* need to be one of the vertex points from step 1.

3. Now, the sequence definition. For each positive integer n, we define p_n to be the *midpoint* between p_{n-1} and a *randomly-chosen vertex point*, where the midpoint's coordinates are rounded **down** to the nearest integer (e.g. in Python using `// 2`).

   For example:

   - To define p_1, we:

     a. Choose a random vertex $v_1 \in VERTICES$, e.g. $v_1 = (700, 100)$.

     b. We set p_1 to be the midpoint of p_0 and v_1:

     $$p_1 = \left(\frac{500 + 700}{2}, \frac{500 + 100}{2}\right) = (600, 300)$$

   - Then to define p_2, we:

a. Choose a random vertex $v_2 \in VERTICES$, e.g. $v_2 = (100, 100)$. *This vertex can be the same as the one chosen for p_1 (i.e., v_2 can equal v_1).*

b. We set p_2 to be the midpoint of p_1 and v_2:

$$p_2 = \left(\frac{600 + 100}{2}, \frac{300 + 100}{2}\right) = (350, 200)$$

Note that Step 3 really defines an *infinite* sequence of points $(p_0, p_1, p_2, \ldots)$. But from a computational perspective, we'll use a for loop to generate a fixed (and finite!) number of points.

2. Your next task is to implement the two functions `generate_point_sequence1` and `draw_point_sequence1`. Both of these two functions will use a for loop to generate a sequence points according to the above definition of "Point Sequence 1", but in different ways: `generate_point_sequence1` will return a list of the points, while `draw_point_sequence1` will draw those points in a Pygame window.

   Here are some sample interactions in the Python console you can use to call `generate_point_sequence1`:

```
>>> vertex_points = [(100, 100), (700, 100), (700, 700), (100, 700)]
>>> initial_point = (500, 500)
>>> generate_point_sequence1(vertex_points, initial_point, 4)
...
```

Testing Point Sequence 1

Now, because of the random choices, you can't test that `generate_point_sequence1` will always return a certain expected value. Instead, you can manually verify that each point in the sequence *could have been generated* by following the definition of Point Sequence 1.

For example, suppose you execute the above code and get the following result:

```
>>> generate_point_sequence1(vertex_points, initial_point, 4)
[(500, 500), (600, 600), (350, 350), (525, 225)]
```

You can then check:

1. That the first point in the sequence equals the initial point, `(500, 500)`.

2. That the second point is *one of the four possible midpoints* between `(500, 500)` and the four `vertex_points`.

   In this case, the four possible midpoints are `(300, 300)`, `(600, 300)`, `(600, 600)`, and `(300, 600)`. The second point in the list is one of them!

3. Repeat Step 2 for each of the remaining points.

Here is an example call for `draw_point_sequence1`.

```
>>> draw_point_sequence1(800, 800, vertex_points, initial_point, 100)
```

With just 100 points, you won't see a pattern emerge, so the window pygame visualization might feel like a bit of a let down. Try changing the number of points to `1000000` instead—you'll see a lo more points fill the screen, but not necessarily a fun pattern.

But, try using these three vertex points, which form an equilateral triangle:

```
>>> vertex_points = [(700, 400), (250, 660), (250, 140)]
```

With a large enough number of points, you should see something pretty spectacular! Try experimenting with different vertex and initial points; you may also choose to use `a3_helpers.regular_polygon_vertices`.

WARNING: it can be tempting to check the correctness of your code by looking at visualizations only. Visualizations may be pretty, but they aren't precise—make sure to check your work in `generate_point_sequence1` carefully, and make sure your code in `draw_point_sequence1` is very similar.

Next, let's learn about a variation of our first point sequence definition that can be used to generate nev (but still very cool) patterns.

Point Sequence 2

This point sequence is the same as Point Sequence 1, except with the following changes:

- There must be *at least four vertex points* (instead of at least three).
- The first three points, p_0, p_1, p_2 are defined in the same way as Point Sequence 1.

- For all postive integers $n \geq 3$, p_n is defined by first choosing a vertex v_n, like Point Sequence 1. **However, how v_n is chosen is different**:
  - If the *same vertex* was selected when choosing p_{n-1} and p_{n-2} (i.e., $v_{n-1} = v_{n-2}$), then v_n is chosen randomly among the vertices *that are NOT v_{n-1} or a neighbour of v_{n-1}*.
  - Otherwise, v_n is chosen randomly among all vertices, like Point Sequence 1.

Note: We define the **neighbours** of a vertex v to be the vertexes that are adjacent to v in the list of vertices, *and* the special case where we consider the first and last vertices in the list to also be neighbours *of each other*. (This is why we're storing the vertices as a Python `list` rather than a `set`!)

3. Your next task is to implement the functions `generate_point_sequence2` and `draw_point_sequence2`, which are similar to the functions from the previous question, but use Point Sequence 2 instead. You can test your functions in the same way as well, and you should see different patterns appear based on what you've written.

4. When you read our description of how to test "Point Sequence 1" above, your alarm bells might have been going off at the phrase "Repeat Step 2 for each of the remaining points."

   That's right! Your next task is to *implement the function* `verify_point_sequence1`, which will take a list of vertex points, an initial point, and a list of points `points`, and check whether `point` could have been generated from the vertex and initial points using the Point Sequence 1 definition. Or in other words, you'll implement a function that performs the "manual testing" we described above.

   Here is how you could use `verify_point_sequence1` to test your work in the Python console:

```
>>> vertex_points = [(100, 100), (700, 100), (700, 700), (100, 700)]
>>> initial_point = (500, 500)
>>> new_points = generate_point_sequence1(vertex_points, initial_point,
        4)
>>> verify_point_sequence1(vertex_points, initial_point, new_points)
True
```

   Notes:

   - This function is probably the most complex task you'll complete on this assignment, so if you're stuck don't be afraid to move onto Question 5 and come back to this later.

- We are *not* requiring that you implement an analogous `verify_point_sequence2`, thoug you certainly can for additional practice (and testing).

5. Now for our big finale! So far your functions have taken the vertex points and initial point as parameters. Your final task for this assignment is to implement `user_pattern`, which will inste use Pygame to let the user click in the Pygame window to select the vertex points and initial point This function will bring together together everything you've done so far in this part of the assignment, as well as your learning about user interactions from Tutorial 5 (https://www.teach.cs.toronto.edu/~csc110y/fall/tutorials/05-loops-and-mutation/).

   At the end, you should have a pretty cool interactive Pygame application that you can show off to your friends and family! 🥳

Acknowledgements

This part of the assignment was inspired by this blog post (https://fronkonstin.com/2019/10/28/the-chaos-game-an-experiment-about-fractals-recursivity-and-creative-coding/).

Submission instructions

Please **proofread** and **test** your work carefully before your final submission! As we explain in Runnin and Testing Your Code Before Submission (https://q.utoronto.ca/courses/278340/pages/running-and testing-your-code-before-submission?module_item_id=3953751), it is essential that your submitted code not contain syntax errors. **Python files that contain syntax errors will receive a grade of on all automated testing components (though they may receive partial or full credit on any TA grading for assignments).** You have lots of time to work on this assignment and check you work (and right-click -> "Run in Python Console"), so please make sure to do this regularly and fix syntax errors right away.

1. Login to MarkUs (https://markus.teach.cs.toronto.edu/2022-09/courses/1).
2. Go to Assignment 3, then the "Submissions" tab.
3. Submit the following files: `a3.tex`, `a3.pdf` (which must be generated from your `a2.tex` file), `a3_part1.py`, `a3_part2.py`, and `a2_part3.py`. Please note that MarkUs is picky with filenames, and so your filenames must match these exactly, including using lowercase letters.

- Please check out the Overleaf tutorial on exporting project files (https://www.overleaf.com/learn/how-to/Exporting_your_work_from_Overleaf) to learn how to download your `a3.tex` and `a3.pdf` files from Overleaf.

4. Refresh the page, and then *download each file* to make sure you submitted the right version.

Remember, you can submit your files multiple times before the due date. So you can aim to submit you work early, and if you find an error or a place to improve before the due date, you can still make your changes and resubmit your work.

After you've submitted your work, please give yourself a well-deserved pat on the back and go take a re or do something fun or look at some dynamical systems or eat some chocolate!

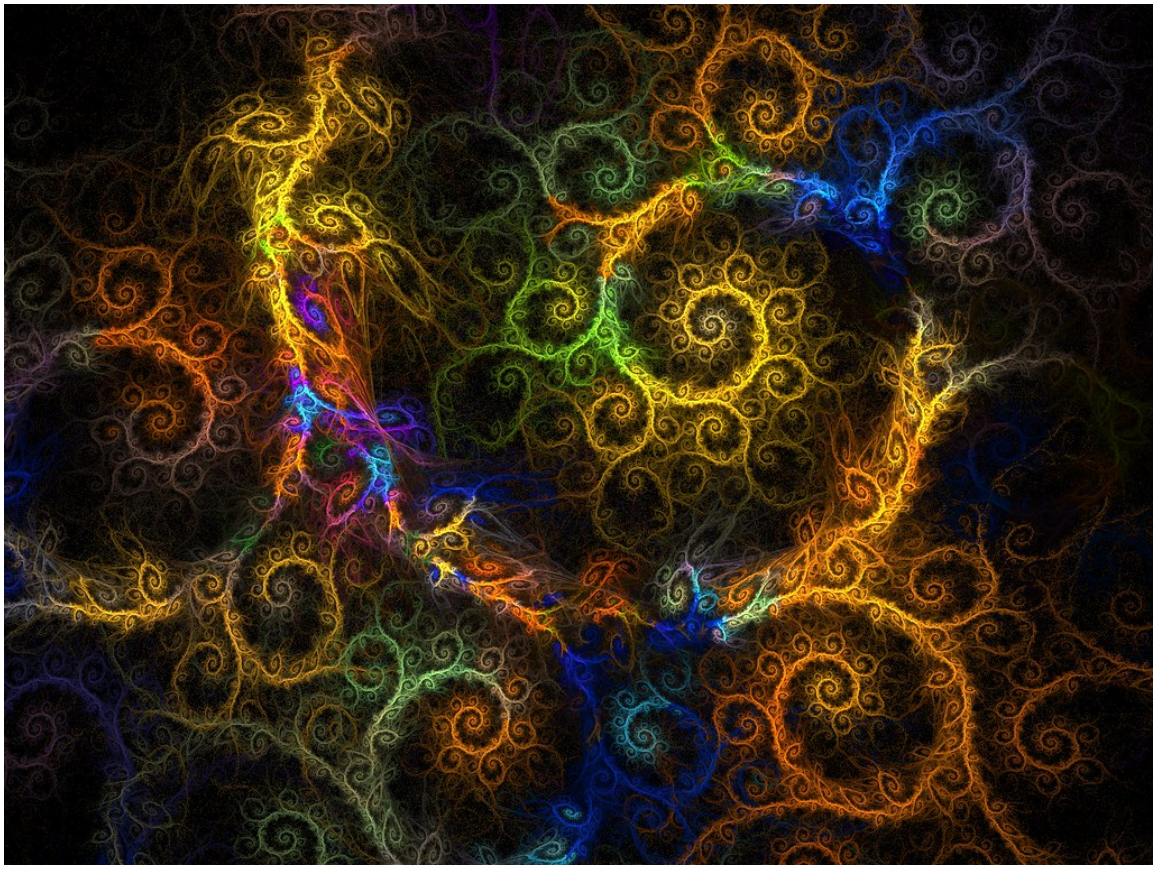


Image credit longan drink (https://www.flickr.com/photos/longan_drink/289130767/sizes/o/)