**CS 251: Scripting and Version Control: Outlab**

- Due: 11:55 AM, 27th August
- Please write (only if true) the honor code. If you used any source (person or thing) explicitly state it. You can find the honor code on Piazza.

## Introduction

In the in-lab you have been introduced to two of the most popular and powerful technologies in Computer Science, the ubiquitous scripting language, Python and the pre-eminent version control system, Git. In this outlab we shall attempt to use these technologies together on a software project. For the purposes of secrecy we will be working with a private repository during the lab. However, once the submission is complete, feel free to put your work up on Github or git.cse and let the world see your code!

## Task A - What should I watch?

In this part, we shall be using Python to construct a movie recommendation system, based on the Collaborative Filtering algorithm. Minor variations of this algorithm are used on most recommendation engines, including those on famous e-shopping and movie rating websites. Whenever you see the words, "People who liked this also liked", you know some form of collaborative filtering is at work. We shall be using Git to version our work. As in the inlab, writing elegant pythonic code is an ideal we wish to encourage. When you use a cool Python feature to save yourself some time or to express yourself more cleanly, do mention it the comments or documentation for a few extra points (and nerd bragging rights).

### 1. Getting Started

Let's start with the basics. Initialize a git repository in a directory `task_A` on your system. Let this remain a private repository for now.
For any recommendation system, one needs data! This data is usually gathered from users on a website. Python provides you several ways to use public APIs to obtain freely available data. For this task, however we shall be providing the data. Download the provided file `movie-ratings.csv` and add it to your repository.
The file contains information on movie ratings in the following manner. The first line of the file contains the following entries separated by commas
*Name,Movie 1, Movie 2,Movie 3,,,, ...*
The next lines contain the corresponding entries (Name and individual movie ratings out of 10) for each independent user/reviewer. If the movie has not been reviewed by a particular user, the corresponding entry in the file would be -1.
Python provides great inbuilt facilities for parsing common data formats (which is part of what makes it a very popular language for data science). Use these facilities to read in the data from this file and organize it into data structures of your choice. Use a script `collab-filter.py` for this purpose.
Next edit the script to take input from the user consisting of his/her name and their review scores for a subset of the movies under consideration. The input format will be another csv file, `user_preference.csv` in the same format without the name field.
At this point, make your first commit. Document your commit message and commit number (figure out how to find that!) in your readme. Now we're set to go further.

### 2. The Distance between Souls (Branching Out and Merging In)

The crux of collaborative filtering is this: ”Reviewers who agree on a significant subset of the material are more likely to be in agreement over the rest.”
To use the technique we need to come up with a way of finding how similar a given reviewer's tastes are to our own. If we consider our scores to the various movies as a vector, then what we are after is a method of computing the distance between two such vectors. There are several methods of such distance computation. Being scientifically minded (we hope!), we would like to try out a couple of such methods before we settle on one. (For this part, document all git commands used in the readme)
**(i) Euclidean Distance:**
Our first notion of distance will be the familiar one of Euclidean distance. Be careful however, all reviewers have not viewed the same movies. Similarities must be only taken over the commonly reviewed subset.
Create a git branch called **euclidean** in your repository. Switch to this branch and edit your python program to sort all the critics by their euclidean distance based similarity to the program user (based on the subset of programs the user has reviewed.) Once, satisfied make a commit on this branch. As before, document the relevant commit message and commit number.
**(ii) Pearson Correlation:**
A second (and generally considered superior) method of finding similarity between preference vectors is known as Pearson Correlation (you may have encountered this in your course on Data Analysis and Interpretation). To find a definition and description of this factor look here. Note that we must again be careful to only calculate the factor over mutually reviewed items.
Now switch back to the master branch of your repository. Create a git branch called **pearson** in your repository. Switch to this branch and edit your python program to sort all the critics by their Pearson correlation based similarity to the program user (based on the subset of programs the user has reviewed.) Once, satisfied make a commit on this branch. As before, document the relevant commit message and commit number.
**(iii) Bringing it all Back Together:**
Now that you are done playing with the two versions of your distance calculation, it's time to use the Pearson Correlation for our further work, as it is the superior measure. (Can you see why this is? Let us know in the readme for some bragging rights!)
Now merge the pearson branch into the master branch. Check the latest commit on the master branch, and state it's message and number on the Readme.
Delete the pearson branch as that chain of development has been successfully completed and incorporated. Leave the euclidean branch as it is, so we can look at your work.

### 3. Humanity's Answer

Now that you have a ranking, with corresponding distance coefficients for your critics, modify your script to calculate a weighted rating for each movie depending on how far away the critics who rated it are, sort the movies on this weighted rating, and suggest 3 unwatched movies for the user to watch, in order. If the user has less than 3 unwatched movies output them all in order of preference.

## Task B: Classes, Ducks and Conflicts

### 1. Everything is an Object!

Python is an object oriented language. In many ways it's object orientedness is even more marked than in other popular languages such as C++ and Java. Everything in Python, including integers and floats are objects, not primitive types. This leads to a very strong notational and stylistic consistency within various definitions. In this section we shall see how to take advantage of this.
Start a new git repository called `taskB`. Create a file called `complexno.py` which contains a class ComplexNumber. This class should provide the basic mathematical functionalities that are present

on real numbers such as addition, multiplication and the ability to be directly printed with a print statement. Study operator overloading in Python to figure out how you make this happen. (You might want to look at the `__str__` and `__add__` functions, to get on the right track). At this point, you should be able to add complex numbers $\mathtt{c}_1$ and $\mathtt{c}_2$ by the command $c_1 + c_2$, and print a complex number $c$ via `print(c)`. `print(c)` on a number with real part $a$ and imaginary part $b$ should display $a + \iota b$ The other arithmetic operations should work the same way.
Now, make a commit and document it.

## 2. If it quacks like a duck...

You must have noticed by now that Python does not require you to declare the types of your variables. How then does it infer whether a particular statement satisfies the requirements of type correctness? This system is known as dynamic type checking, or more colloquially, duck typing. Read up on duck typing and make sure you understand how it works.
Open a python repl. Import the complex number module you created earlier. Try to sum a series of integers, floats, strings and complex numbers using the sum function. Does the sum function work on complex numbers? Why? Write an explanation in your Readme.
Now, create a new file, `trytypes.py`. Import the complex number module you created earlier. Write a function myexponent which given an argument x calculates $e^x$ using the infinite series retaining terms upto the $x^6$. This function should work unchanged for integers, floats and complex numbers.

## 3. Working together

We are now going to add a couple more methods to our complex number class. However as is often the case within large organizations these functions shall be added on separate branches of development.
Create a branch called `representation`. Here add a `__repr__` method to your complex number class. Make a commit.
Now switch back to the master branch, and add a method to your complex number class which returns another complex number containing the square of the original. Call this method `mysquare`. Make a commit on the master branch.
Note that the newly created function does not exist on the `representation` branch. Now, merge in the representation branch. Does the branch succeed and does your code now work properly with both functions? Once done, delete the `representation` branch.

## 4. *Extra Credit:* Conflict is unavoidable

So far you've been committing and merging happily with no discord or friction. But good times must come to an end. A bitter disagreement starts with regard to the `__str__` function for the complex numbers. One of the team members, being into electrical engineering whats complex numbers printed as $a + j.b$. The other, an algebra nut, prefers the general (a,b) notation.
Create a branch called `electricity!` and change the `__str__` method to reflect the electrical engineers display style of choice. Commit on this branch.
Now, switch to the master branch and change the display style to the algebraist's choice. Commit on this branch.
Now try to merge `electricity!` into master. Does a conflict occur? List the steps taken to fix this merge conflict. Take a screenshot of yourself doing this and add it to your submission. While fixing the conflict you can preserve whichever style you like better .(I have a preference for the algebraic one). In large projects such conflicts can occur on a large scale. using branches allows them to be safely resolved without necessarily corrupting the production / master branch.

## Submission Guidelines

1. When you submit, please document individual percentages such as Student 1: 80%, Student 2:100%, Student 3:10%. In this example, the second student will get full marks (10/10) and the first student will receive 8/10.
2. Do include a readme.txt (telling me whatever you want to tell me). (The reflection essay is not required for inlab, but is required for outlab.) Do include group members (name, roll number), group number, honour code, citations etc.
3. The submission folder should also contain two folders `taskA` and `taskB`, which are the git repositories you created earlier for the tasks. These repositories contain your code (named as specified in the tasks), and must retain the git branches we have asked you to preserve.
4. The folder and its compressed version should both be named `lab04_groupXY_outlab` for example folder should be named `lab04_group07_outlab` and the related `tar.gz` should be named `lab04_group07_oulab.tar.gz`

## How We will Grade You - 70 + 10 Marks

1. Task A.1 - 7 Marks
   (a) Parsing `movie-ratings.csv` using python - 5 Marks
   (b) Parsing `user_preferences.csv` using python - 2 Marks
2. Task A.2 - 20 Marks
   (a) Branching out for euclidean distance - 5 Marks
   (b) Implementing euclidean distance in python - 5 Marks
   (c) Branching out for Pearson correlation - 1 Mark
   (d) Implementing Pearson correlation - 5 Marks
   (e) Merging the Pearson correlation branch with master - 4 Marks
3. Task A.3 - 13 Marks
   (a) Calculation of a weighted rating - 9 Marks
   (b) Sorting and suggestion of movies - 4 Marks
4. Task B.1 - 12 Marks
   (a) Creating the repo `taskB` - 2 Marks
   (b) Arithmetic operations in ComplexNumber class - 7 Marks
   (c) Printing ComplexNumbers - 3 Marks
5. Task B.2 - 8 Marks
   (a) Explanation of complex number addition - 2 Marks
   (b) Calculation of $e^x$ for integer,floats - 2 Marks
   (c) Calculation of $e^x$ for complex numbers - 4 Marks
6. Task B.3 - 10 Marks
   (a) Creating the representation branch - 1 Mark
   (b) Implementing the `__repr__` method in the correct branch - 2 Marks

   (c) Implementing `mysquare` in the correct branch - 2 Marks
   (d) Merging the representation branch - 5 Marks

7. Task B.4 - 10 Marks
   (a) `electricty!` branch with modified `__str__` - 2 Marks
   (b) Steps taken to merge the conflict - 6 Marks
   (c) Screenshot of fixing the merge conflict - 2 Marks