The University of Texas
Computer Science

313E Programming Assignment 13
Registration Planning

Setup

This assignment may be completed individually or with a pair programming partner. Be sure to include your and your partner's name and EID in the Python file header. If you are working alone, you may delete one of them.

Complete these steps to prepare for the Assignment.

Check	Description
▢	Download registration.py, registration.in, small.in, eoc.in, simple.in, and cycle.in. You will be working on the registration.py file. You may not change the file names. Otherwise, the grading script will not work.
▢	Place all files in the same folder/directory.

If you would like to turn on autosave in VSCode, click on **File -> Autosave.**

Problem Description

For this programming assignment, you will be writing a program to complete a registration plan. College courses often mandate certain prerequisites for enrollment. You will be determining a valid sequence of courses for a student to enroll in to ensure completion of all required courses.

Requirements

Given a directed graph representing UT courses, determine if the graph is cyclical. A cyclical graph means that there is no valid registration plan, and examples are provided further below. If the graph is acyclical, then find a valid registration plan. We have defined the **Stack, Queue,** and **Vertex** classes for you. Based on these classes, you will be working on methods in the **Graph** class. You may add as many helper functions as needed.

There are a few restrictions you must follow:

1. Your program uses the provided text (.in) files to create your final output. You will be using input redirection to read from these files.
2. You **must** implement your solution using Graphs and the methods we provide.
3. **You may not change the names or parameters of the functions listed. They must have the functionality as given in the specifications. You can always add more functions than those listed.**
4. **You may not import any additional external libraries in your solution besides sys, os, and deque from collections.**

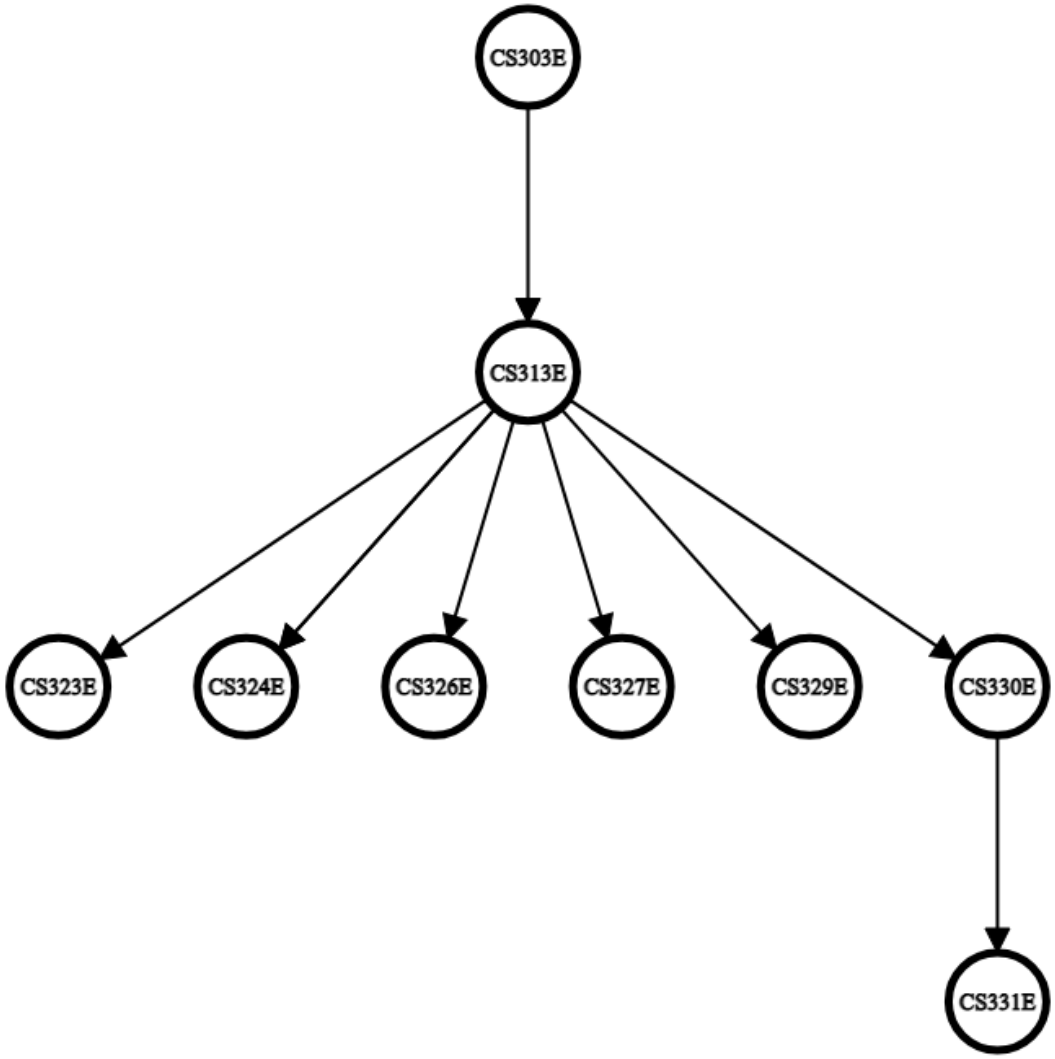
Using the provided .**in** files as input, you will modify the following methods. Further details breaking down the input files are listed in the **Input** section.

def has_cycle(self)
Determine if the graph has a cycle. If a cycle exists, return true. Otherwise, return false. This method **must** be implemented using a depth-first search implementation. A recursive approach is **highly recommended** (using a helper function), although you are allowed to implement it iteratively using a stack.

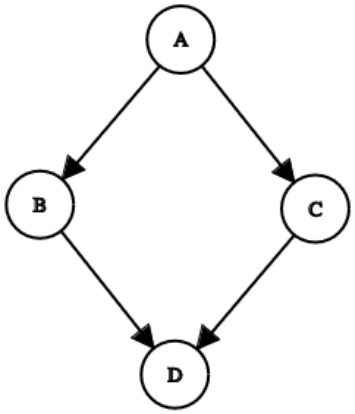
def get_registration_plan(self)
Return a valid registration plan as a list of vertex labels. The registration plan should be the order in which the courses should be taken. This method may safely assume that there is a valid registration plan.

def main()
Read the input file and construct the graph. The output code has been written for you.

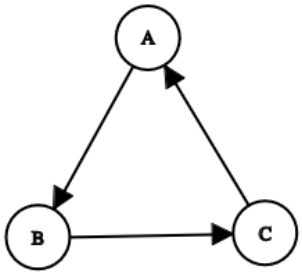
The images below are examples of graphs representing classes for registration.



In the image above, we demonstrate an example of a graph representing classes that can be taken for the Elements of Computing certificate. We can see the prerequisites required for taking the courses. CS303E does not require a prerequisite, but the prerequisite for taking CS313E is to complete CS303E. The only other class that requires a prerequisite is CS331E. In this example, there are no cycles, and a valid registration plan can be found. The input file **eoc.in** represents this graph.



In this image, we represent a simple example of required class prerequisites with a graph. Class A has no prerequisites. Class B and C require class A, and class D requires class B or class C. A valid registration plan can be found. The input file **simple.in** represents this graph.



In this example, we represent required class prerequisites with a simple graph. However, in this graph is a cycle, meaning that it is impossible to build a valid registration plan. Class B requires class A, and class C requires class B. However, class A can only be taken after taking class C, demonstrating an impossible registration plan. The input file **cycle.in** represents this graph.

As a hint for cycle detection, try to identify **back edges**. Suppose we were doing DFS. A **back edge** is an edge that connects to any vertex that is already in the current DFS path. If we started from Vertex A, when processing Vertex C, our call stack (or stack if you choose to do it iteratively) should have ABC (with C at the top). This is the current path being explored by DFS. If there is an edge that leads to any of the vertices already in this path, we have found a back edge and therefore a cycle. Since Vertex C has an edge to Vertex A, which is already in the current path, then a cycle has been detected.

Input

Using **registration.in** as an example, each input file contains vertices and edges of a directed graph using this structure:

Input File Content	Description
`14`	The number of vertices.

m n o p q r s t u v w x y z	Each line contains the label for one vertex. The labels are unique, the vertices are 0-indexed (as in, the vertex with label a should be at the 0th index in the list of vertices).
21	The number of edges.
m q m r m x n o n q n u o r o s o v p o p s p z q t r u r y s r u t v w v x w z y v	Each line contains one directed edge. The structure for each edge is: from_vertex, a space, then the to_vertex.

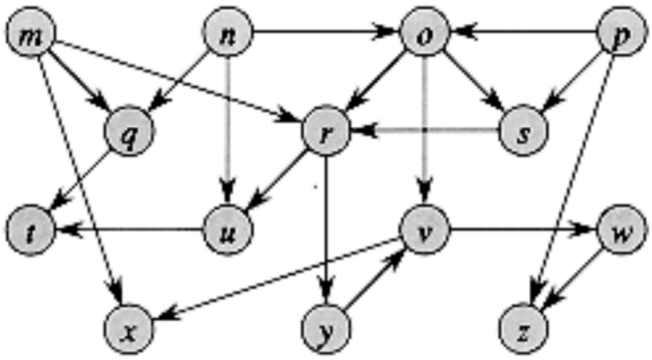
The starter code uses this implementation for the graph:

- vertices - A list of vertex objects, each with a label and visited attribute.
- adjacency_matrix - An adjacency representation of the graph

Using **registration.in**, the adjacency matrix is as follows:
(The vertex labels are included for understanding. They are not part of the matrix.)

```
  m n o p q r s t u v w x y z
m 0 0 0 0 1 1 0 0 0 0 0 1 0 0
n 0 0 1 0 1 0 0 0 1 0 0 0 0 0
o 0 0 0 0 0 1 1 0 0 1 0 0 0 0
p 0 0 1 0 0 0 1 0 0 0 0 0 0 1 p's outgoing edges
q 0 0 0 0 0 0 0 1 0 0 0 0 0 0
r 0 0 0 0 0 0 0 0 1 0 0 0 1 0
s 0 0 0 0 0 1 0 0 0 0 0 0 0 0
t 0 0 0 0 0 0 0 0 0 0 0 0 0 0
u 0 0 0 0 0 0 0 1 0 0 0 0 0 0
v 0 0 0 0 0 0 0 0 0 0 1 1 0 0
w 0 0 0 0 0 0 0 0 0 0 0 0 0 1
x 0 0 0 0 0 0 0 0 0 0 0 0 0 0
y 0 0 0 0 0 0 0 0 0 0 0 0 0 0
z 0 0 0 0 0 0 0 0 0 0 0 0 0 0
      ^--- p's incoming edges
```

For your visual aid, **registration.in** is represented by the image below.



Output

Note that there may be multiple possible solutions for a registration plan. The grading script will test the order instead of comparing the text output.

For each input file provided, here are possible outputs as follows:

Input File	possible get_registration_plan() return value
registration.in	`['m', 'n', 'p', 'o', 'q', 's', 'r', 'u', 'y', 't', 'v', 'w', 'x', 'z']`
eoc.in	`['CS303E', 'CS313E', 'CS323E', 'CS324E', 'CS326E', 'CS327E', 'CS329E', 'CS330E', 'CS331E']`
small.in	`['a', 'b', 'e', 'c', 'd']`

simple.in	`['a', 'b', 'c', 'd']`

As an example of the complete output, here is the output for small.in:

```
Valid registration plan detected.

Registration plan:
['a', 'b', 'e', 'c', 'd']
```

Here is an example output for cycle.in:

```
Registration plan invalid because a cycle was detected.
```

Running Your Program

You can test your program using the **.in** files we have provided for you. You can run the program by typing the following command into your terminal, replacing the **.in** file with the desired input file:

```
python3 registration.py < registration.in
```

Alternatively, if you are using a Windows computer and are having trouble with input redirection <, you can run:

```
cat registration.in | python3 registration.py
```

Grading

Visible Test Cases - 75/100 points
The program output exactly matches the expected output for each function. To receive full credit, the program must produce the correct output based on all requirements in this document and pass all the test cases.

Hidden Test Cases - 15/100 points
The program output exactly matches the expected output for each function. To receive full credit, the program must produce the correct output based on all requirements in this document and pass all the test cases.

Style Grading - 10/100 points
The registration.py file must comply with the PEP 8 Style Guide.

If you are confused about how to comply with the style guide, paste the error or the error ID (e.g. C0116 for missing function or method docstring) in the search bar here in the Pylint Documentation, and you should find an example for how to fix your problem.

The TAs will complete a manual code review for each assignment to confirm that you have followed the requirements and directions on this document. Deductions will occur on each test case that fails to follow requirements.

Submission

Follow these steps for submission.

Check	Description
▢	Verify that you have no debugging statements left in your code.
▢	Submit **only** registration.py (the starter code file with your updates) to the given assignment in Gradescope. This will cause the grading scripts to run.
▢	If you have a partner, make sure you are submitting it for you **and** your partner on Gradescope. See this Gradescope documentation for more details.

When the grading scripts are complete, check the results. If there are errors, evaluate the script feedback, work on the fix, test the fix, and then re-submit the file to Gradescope. You can submit as many times as necessary before the due date.

NOTE: By default, only the most recent submission will be considered for grading. If you want to use a previous submission for your final grade, you must activate it from your submission history before the due date.

Academic Integrity

Please review the Academic Integrity section of the syllabus. We will be using plagiarism checkers, and we will cross-reference your solution with AI-generated solutions to check for similarities. Remember the goal in this class is not to write the perfect solution; we already have many of those! The goal is to learn how to problem solve, so:

- Don't hesitate to ask for guidance from the instructional staff.
- Discuss with peers about **only** bugs, their potential fixes, **high-level** design decisions, and project clarifying questions. Do not look at, verbalize, or copy another student's code when doing so.

Attribution

Thanks to Dr. Carol Ramsey and Dr. Kia Teymourian for the instructions template and this assignment.