MCS 260 – Introduction to Computer Science – Fall 2020 – David Dumas

Project 4 – Proposal due Nov 16 – Project due Dec 4 at 6pm Central

1. Overview

Projects 1–3 in MCS 260 asked you to write programs that followed a precise specification, which were primarily evaluated by an autograder.

Project 4 is different: It is an open-ended programming project where you have a lot of flexibility to determine what kind of program you'd like to write. It is important that whatever you choose is a program of significant complexity that will use multiple skills from MCS 260.

Project 4 will be collected in Gradescope, which will perform some basic checks to make sure the filenames meet the specifications given below. Manual review will account for the majority of the points on this assignment. Specifically, there will be 25 points available for this project, with automated checks accounting for 3 points, and the remaining 22 points being determined by manual review of your program.

2. Proposing a project

The sooner you decide on your project topic, the sooner you can start working, and the more time you will have to complete it. The deadline for submitting a project proposal is **November 16**.

Project proposals must be submitted using this online form:

https://forms.gle/7dK4forfBY1U3kzs9

— BUT DON'T SUBMIT THE FORM UNTIL YOU HAVE READ THIS PROJECT DESCRIPTION —

To access the form, you will need to be logged in with your `netid@uic.edu` Google account.

The form asks you several questions about your project. In particular it asks for a:

- title and short description of your project
- list of some concepts from MCS 260 that will be used in your program (e.g. object-oriented programming, modules, regular expressions, recursion, ...)
- description of how I will test your program (e.g. Do I just run one Python file? Are there input data files I need?)

Within a day or two of submitting your project proposal, you will receive email notification either approving the proposal or requesting a revision of it (e.g. additional information). If your initial proposal results in a request for revision, it is important to act promptly to submit a revised version so that you maximize the time you have to work on the actual project.

3. General requirements

3.1. **Must use course topics.** Your project must be designed to involve some of the topics from MCS 260, and a list of such topics must be included with your proposal.

3.2. **Must be testable.** With manual testing of many student projects by a single instructor, it is important that the process of testing of your project is straightforward and clearly explained. Testing must not involve the creation of custom data files or other time-consuming preliminary steps. If your program requires these, sample input files should be included with your submission.

3.3. **Must be documented.** The next section describes requirements for the source code of your project. In addition to those requirements, your project must include a file `README.txt` that describes the project in at least as much detail as the proposal, and which documents how it should be tested. This last point is extremely important:

If `README.txt` is missing, or if I read `README.txt` and cannot figure out how to use your program rather quickly, there will be a significant grading penalty.

A good litmus test for whether your `README.txt` is good would be: If you gave your project to someone else in the course, could they figure out how to use it just by reading `README.txt`? It would be a good idea to actually try this! (Unlike previous projects where everyone was working on a similar task, in this case, asking someone else to try your program is acceptable.)

It is acceptable to include additional documentation, such as a tutorial or reference manual.

You must be the sole author of the documentation. It should not consist primarily of quotations from other sources, and any quotations present should be cited.

3.4. **Must not depend on special characteristics of your computer.** It must be possible to test your project using Python 3.6 on Windows 10 or Ubuntu Linux 20.04. If you know that your program requires one of these platforms (and doesn't work on the other), that should be indicated in the `README.txt` file. The program's operation must not depend on any system-specific details such as the exact operating system version, the user name, or the name of the directory in which the program is run.

(If you develop on Mac OS X, it is exceedingly likely that your program will run on Ubuntu Linux 20.04 with no changes.)

3.5. **Must only modify files in current directory.** Your program must not alter or write any files outside of the current working directory when the program is run, unless a filename is provided as input to your program that specifies another location.

3.6. **No external modules without approval.** Your program may not import any modules other than those included with the Python standard library unless this is indicated in your proposal and approved.

3.7. **Not a standard exercise.** Some programming projects (e.g. write a tic-tac-toe game) are so often assigned in introductory courses that hundreds of solutions and lots of related discussion can be found online. If you propose one of these "standard" topics, you will be asked to choose another topic or revise your proposal significantly. Basically, if an idea shows up prominently in a web search for something like "Python beginner project ideas", it will probably not be acceptable.

3.8. **Limited overlap with other students.** If your proposal is substantially similar to that of another student in MCS 260 who submitted their proposal before you submitted yours, you will be asked to make changes. The goal is for each student to have a unique project topic.

Note that many of the sample topics listed below allow for many different proposals to be derived from them, depending on how you fill in the details.

3.9. **Limited file size.** The total size of the submission to Gradescope must not exceed 10MiB. If there are larger data files relevant to your project, they should be publicly available on the internet, and you should include instructions for how to obtain them in the project documentation.

3.10. **We allow mostly-documentation projects focused on a new topic.** You can propose a project that focuses primarily on learning a new topic in Python programming, for example, a module from the standard library that we don't cover in MCS 260. If this is your plan, you should indicate this in the proposal form, and when you submit your project it should include:

- A description in about 300-500 words of what you learned about the module/feature in `README.txt`.
- A text file `EXERCISES.txt` that contains at least three programming exercises that would be appropriate to assign to someone learning this topic.
- A main program `solutions.py` that contains solutions to the exercises in a form that allows all of them to be run, and which reports on the results by printing to the terminal.

4. Source code specification

This section describes how your program must be written and submitted.

Your submission must contain a `README.txt` file meeting the requirements of Section 3.3.

Your submission must contain at least one Python source file. All Python source files in your project must have the extension `.py`.

You may choose the name of your main program, and of any modules you create in the project.

Each Python source file you submit must begin with a standard header, which consists of a file-level docstring followed by:

- The following comment line, verbatim: `# MCS 260 Fall 2020 Project 4`
- A comment line consisting of your full name
- A comment line beginning with `# Declaration:`, and then containing a full sentence that says, in your own words, that you are the sole author of the program you are submitting and that you followed the rules from the course syllabus in preparing it. (This part can span multiple lines.)

Here is an example of an acceptable start of a source file for a project submitted by a student named Srinivasa Ramanujan:

```
"""Virtual machine that runs 6502 assembly code"""
# MCS 260 Fall 2020 Project 4
# Srinivasa Ramanujan
# Declaration: I, Srinivasa Ramanujan, am the sole author of this code, which
# was developed in accordance with the rules in the course syllabus.
```

Your source code will be evaluated on the basis of readability. You are expected to use descriptive variable names for the most important variables (e.g. `credit` as opposed to `c`). Single-letter names are permissible for the variable of a for loop.

Comments should be included whenever the intent of a line is not immediately apparent. Comments on every line would be excessive, but it is expected that this project will require many descriptive comments. Judging the correct comment density involves an element of subjectivity. If you are unsure about whether your code has enough comments, just ask. (You can ask about this by submitting your code to Gradescope and then emailing the instructor, or by including source code directly in an email to the instructor.)

Every comment in the source files you submit should consist of explanatory text. Do not use comments to disable code that isn't used; instead, remove such code before submitting.

In the review of your source code, my ability to understand the way your code works is important. It is theoretically possible to write an extremely complicated program that performs a simple task correctly but which is impossible for a human to understand. In the code review scoring, code that is very difficult to understand may be subject to a penalty. If you are unsure about the level of understandability of your code, ask.

5. No collaboration or use of external resources

As with every other project in MCS 260, Project 4 is subject to the rules set out in the syllabus. This means that any program you propose to create for Project 4 must be entirely your own work, and you must not give or accept assistance of any kind except from course staff. In particular, using code adapted from online resources is forbidden. It **is** acceptable to use books and online resources to learn new skills and techniques in this project, especially if you propose a project involving topics not covered in MCS 260. Learning how to do something is quite different from finding code that does something and pasting it into your project, and these are generally easy to distinguish when reviewing submissions. But if you are at all unsure about whether a certain way of using a resource is permitted or not, ask before proceeding.

Violations of the rules in the MCS 260 syllabus will be referred to the Dean of Students office for disciplinary proceedings.

6. Project seeds

Here is a list of some "seeds" for projects that you could expand and develop into a proposal. You are free (and encouraged!) to pursue ideas not on this list. You are also free to modify any aspect of one of these ideas when you create your proposal (as long as it meets the rules in Section 3).

(1) **Recipe multiplier / unit converter** — A program that reads the ingredients for a cooking recipe from a text file, with lines like `"11 oz powdered bhut jolokia chilies"` or `"1.5 c butter"` and then, upon request, does any of these things: (1) Convert from US/imperial units (cup, ounce, pound,...) to metric units (ml, g, ...), (2) Convert from metric to US/imperial units, (3) Multiply the recipe quantities by a given float (e.g. double or halve). Ideally this would use the `QuantityWithUnit` class developed in worksheet 9.

(2) **Interactive fiction** — A game based on keyboard input that prints a description of your current location and lets you type commands like "north" to walk north, "get laser" to pick up an object called "laser", "open tank" to release the sharks, etc.. It should be possible to win the game by achieving a certain objective, such as reaching a specified location, finding a specified object, or something else.

(3) **Filing system** — A program to catalog a collection (e.g. of books), storing its data in a file (e.g. CSV). The program would use a keyboard interface and offer commands to locate (search), print, modify, and delete records. The search command should allow searching for text contained in a specified field, e.g. find all books whose title contains "shark".

(4) **Expanded text to HTML** — A utility to convert text to HTML that is similar to Project 3, but which allows nested lists. A line beginning with `@@` would represent a second-level list item, and `@@@` a third-level list item, etc..

(5) **Todo list** — A program that maintains a list of tasks with optional due dates. It can display tasks in the order added, or in order of how soon they are due. It supports adding or removing tasks, or changing the deadline for a task. The task list is stored in a file. (The handling of dates and times in this project will be one of the trickier parts.)

(6) **Open data analysis** — Use public data sets from a source such as the Chicago Data Portal, the Illinois Data Portal, data.gov, or a federal agency. Write programs to extract useful information from these data sets and display or save it in a convenient format. Document exactly where you downloaded the datasets, and what commands are used to process each one. Example: List the street names of Chicago in order of decreasing numbers of parking tickets issued in 2018. Document what you learned from your analysis of the data.

(7) **Random text** — The program accepts an input text file which should be a very large document written in a natural language (e.g. English). A novel from Project Gutenberg would be a good start. It then produces a bunch of "random" text where the likelihood of any two words appearing next to each other is similar to the input file (but where the actual words are selected randomly). This output text will be meaningless, but at a quick glance it might look like real prose. (This project requires some learning or knowledge of probability and statistics.)

(8) **Python source stats** — A program that reads a single Python source file and prints some statistics about the contents, including the number of lines that are not blank and not entirely comments (true "source lines"), the number of function definitions, number of function calls, the average and maximum line length, etc.. You could use the regular expression module to search for function definitions, or learn about parsing Python source in Python using the `ast` module.

(9) **Simple spell checker** — Find and use a public domain English word list to make a simple spell checker. It should read a given text file (whose name is a command line argument) and print any lines that appear to contain misspelled words. It will then prompt the user to type a replacement word, or let them press Enter to keep the current spelling. The result should be written to a new file. Work to make the program smart about detecting what a "word" is, so that it ignores punctuation and mathematical expressions.

(10) **Find large files** — Recursively search through a directory specified on the command line for files and then print a list of all files found in order of increasing size. For each one, also indicate a percentage of the total size of all files found that it accounts for. Also report on all the directories found in the search, indicating the total size of each directory's contents (the sum of sizes of all files it contains), with directories listed in order of increasing size. Also allow the user to specify a minimum size (e.g. as a second command line argument) so that no items smaller than this theshold are printed. Read about `os.stat()` to learn how to get the size of a file.

(11) **Chase simulation** — Create a program that can simulate two types of robots that move on a 30x30 square grid. Chaser robots can move one square in each time step. A single Target robot can move two squares in each time step. Each Chaser knows the location of the Target robot. The Target robot knows the location of all the Chasers. Program each robot type with a strategy to either seek the Target robot (for Chasers) or to evade Chasers (for the Target). When run, the program shows a diagram of the current state by printing a grid of characters to the screen. Pressing Enter advances time and prints the new state. The simulation ends if the Target is caught by a Chaser.

(12) **Language flashcards** — A program for learning vocabulary in another language that reads a user-supplied data file of corresponding word pairs (with lines like "Frühstück=breakfast") and then selects words at random to ask the user to translate. The user's success rate on each word is tracked and saved to a file. The random word selection should not use a uniform distribution. Instead, it should favor words the user previously got wrong and de-emphasize words the user has a high success rate with.

(13) **Net present value calculator** — Build a tool for analyzing projected profit/loss data for a company taking into account inflation. On the command line, the user provides an input file name and a list of estimated rates of inflation for the next few years. The file should contain a list of estimated profits (positive numbers) or losses (negative numbers), with each line corresponding to one year. The program then computes and prints the net present value of the income stream. (Note: Requires knowledge or learning of the concept of net present value!)

(14) **Personal expense tracker** — A simple program for tracking personal income and expenses. It stores data in a JSON file, and when run prompts the user for commands. Commands can specify spending (with amount, category, and optional date) or income to be recorded, or can ask for a summary of spending and income for the last month, year, or other time period. Such a summary would indicate spending in each category and total spending, each in absolute numbers and as a fraction of income for the same period. (Note: Handling dates/times will be a tricky part of this project.)

(15) **Fake code typist** — A program that lets an actor who doesn't know Python appear to be typing Python code in the terminal for cinematic purposes. First, learn how to detect individual key presses in the terminal in Windows or Linux. Then, make a program that will allow the user to type garbage text (random letters) on the keyboard and have nice Python source code show up on the screen. One character of source code should appear for each key pressed. The code displayed should come from some pre-loaded sample code files that are written by you, are provided as examples in MCS 260, or are in the public domain.

(16) **Location guessing game** — A one-player game displays a grid using text printed to the terminal. Each square in the grid might contain an obstacle (a sinkhole, an enemy, etc.) and the user is allowed to test locations by entering coordinates. The game provides some feedback about each guess, such as the distance to an obstacle, the number of nearby obstacles, or something else. The next time the grid is printed, it also reflects what the player has learned about the contents of the grid squares. You can determine the best win/loss logic, based on what turns out to be fun; examples: (1) Must locate an obstacle within a fixed number of moves to win; (2) Must avoid all obstacles and find all other squares to win; (3) After a limited number of moves, the number of obstacles found is the score.

(17) **Text to LaTeX** — This project is suitable for students who know (or want to learn) the LaTeX document preparation language. It consists of a program similar to Project 3, but instead of converting text to HTML, it converts text to LaTeX with support for variable substitution, comments, and bullet lists.

7. Factors considered in review

This section describes the main factors that will influence how your project is graded.

7.1. **Can I figure out what to do?** By reading `README.txt`, can I easily determine how to use your program and what it does?

7.2. **Does it work?** Does the program do what is promised in the documentation?

7.3. **Does it follow the rules?** Section 3 and Section 4 contain a number of rules that apply to this assignment. A submission that violates any of these rules will not receive full credit.

7.4. **Topic coverage (and bonus).** A project that uses a lot of concepts from the course, especially ones covered after the due date of Project 2, will be viewed favorably. This means that the rest of the project doesn't need to be as complex or polished to earn full credit.

At a minimum, the project must use the concepts in the approved proposal in order to get full credit.

7.5. **Source readability.** Is the source code well-organized, with descriptive variable names, comments, and a docstring for every file, class, and function?

7.6. **Documentation readability.** Is the documentation written in full sentences? Does it make sense, and give a useful summary of what the project is about and how the included Python code can be used?

7.7. **Extra documentation bonus.** As mentioned earlier, some projects may focus primarily on documentation and exercises about a topic we didn't cover in MCS 260.

But for other projects that focus on writing a program, if the project includes a lot of high-quality documentation (beyond just the basic `README.txt` that is required), then I will allow the rest of the project to be less complex and still earn full credit.

7.8. **Use of advanced features bonus.** There are certain concepts in MCS 260 that beginners sometimes shy away from. If you use these extensively, then I will allow the rest of the project to be less complex or less polished and still earn full credit.

Making extensive and effective use of these concepts will have a large effect:

- Object-oriented programming
- Organization into modules
- Tests included with the source (e.g. using pytest)

Making extensive and effective use of these concepts will have a moderate effect:

- Regular expressions
- List and dictionary comprehensions

7.9. **Originality bonus.** A project idea that is not based on one of the suggestions in this document, and not similar to a standard programming example, will be rewarded by allowing the rest of the project to be less complex and still earn full credit.

7.10. **Resuable module bonus.** A project that includes a module containing at least 4 functions that, as a group, seem like they would probably be useful in other programs will be rewarded by allowing the rest of the project to be less complex and still earn full credit.

7.11. **Bad practice penalty.** If your program includes the construct `range(len(...))` in a place where there is a better alternative, there will be a small grading penalty. (Remember, there is almost always a better alternative!)

8. Format of review

You will receive a numeric score for your project and a brief narrative summary of my impressions of it. Both of these will be made available in Gradescope on or before December 16, 2020.

9. Revision history

- 2020-11-06 Initial publication.