README.md

CSCI5801_Group9_P4_ImplementationAndTesting

A shared git repo for group 9 Project Part 4 **and part 5** in the fall 2022 offering of CSCI5801 at the University of Minnesota, Twin Cities.

Files

Included on the GitHub:

Root:

main.py = Used for product evidence, previously used for debugging (still is moderately).

source_and_representations.py = Contains classes for the FileData as well as the associated methods/functions.

class_types.py = new for P5, houses interface for mapping to correct question types, as well as generate and output functionality for the three question types.

multiple_choice.py = Multiple choice generation, printing outputs, etc. was moved from class_types.py to here in order to facilitate development and avoid merge conflicts.

README = .pdf is included for Canvas submission, .md is used by GITHUB. NOTE = GITHUB Repo has been shared with TAs as collaborators, availble here.

test_for_FileData.py = Unit tests for the FileData class.

test_for_ReorderQuestion.py = Unit tests for the reorderQuestion class.

~~end_to_end_test.py = Script for running the end to end test, as defined in the "Executing the Code" section, as well as the Test Execution Report.~~

end_to_end_test.py was not built upon for P5.

test:

test_for_FileData.py = Test code for testing the FileData object housed in sources_and_representations.py.

test_for_MultipleChoice.py = Test code for testing the MultipleChoice object currently housed in multiple_choice.py.

test_for_ReorderQuestion.py = Test code for testing the ReorderQuestion object housed in class_types.py.

test_for_fillinBlank.py = Test code for testing the FillInBlank object housed in class_types.py.

test_files:

Directory of various test files, including files with various programming lanuages file types. Main test file used from here has been testCodeFile.py. When interacting with main.py, files must be within this directory.

Other files included with the submission:

General Files:

Group 9 - Annotation Syntax Tracking = Shows the syntax for annotations used in P4 and those added for implementing P5.

Group 9 - Product Backlogs = 2 Product backlogs created at the beginning of the sprint to guide new development.

Group 9 - Sprint Backlogs = Sprint backlogs derived from the product backlogs, as well as a few geared towards other necessary changes.

Group 9 - Design Document v0.3 = Updated version of the design document to reflect structural and nomenclature changes to the design.

Group 9 - P2 PPALMS Requirements and Use Cases v1.3 = Updated version of the SRS document to reflect structural and use case based changes to the design.

Group 9 - Test Execution Report v0.2 = Report containing the results for the unit testing of version 0.1 of the system and those built out for version 0.2 of the system.

output.txt = An example output file to facilitate product evidence. Generated using testCodeFile.py as input, and generating 5 questions for all types of each grouping.

productEvidence = Video evidence of a teammates roommate functioning as the "user" and generating a file of the same format as the included output.txt file.

Meeting Notes Folder

Includes meeting notes for each of the 6 required meetings for P5 of the project. These include the sprint planning, the three scrums, the sprint review, and the sprint retrospective.

Building the Code

All of the files listed above should be included in the same zipped directory. The directory should be pullled down and unzipped in a convenient location for the user, which has access to Python either through a path or root. The implementation was done using Python 3, so it should run on lab machines out of the box with little set up. Additionally, since this is built on an interpreter, there is little build time, at the expense of run time. Lastly, to try to simplify the process for the TAs, they have been added to the private github repostiory as collaborators.

Executing the Code

In order to execute the code, navigate to the directory within the terminal/command line. If you would like to test the functionality of the system, main.py has been adjusted to allow a simple implementation of a user interface. NOTE: Steps 1 and 2 pertain to downloading the CANVAS submission, the subsequent steps should be the same for GITHUB clone or CANVAS download.

1. Download the zipped folder “Group 9 - P5 Sprint”
2. Unzip the folder in a convenient location with access to python.
3. Open up the command line / Terminal, and navigate to the unzipped/cloned folder.
4. To test, run the following line in the terminal.

```
<python or python3> main.py
```

5. The program will prompt the user to input a filename for testing. The recommended file is the testCodeFile.py file. Enter in this file when prompted. (NOTE: The code automatically searches within the test_files directory.) The testCodeFile.py includes annotations for using 2 groupings of code for reordering questions, a single line "grouping" of code for multiple choice question generation, and a few lines of code grouping for generating fill in the blank questions.

6. main.py will create a FileData object. The amount of identified question groupings is printed to the terminal. The user is then prompted for how many questions they would like to generate based on each grouping type. Generated questions are captured in the generated output.txt file. NOTE: This file is automattically overwritten each time.

   a. For reordering questions: The question number within the grouping, the correct order of the lines, a randomly generated order of the lines, and any lines which order does not matter for are added to output.txt

   b. For multiple choice questions: The question number within the grouping, possible answers, and the correct answer are output to output.txt.

   c. For fill in the blank questions: The question number within the grouping, the unaltered correct "blank" line, any context (code) before the blank, the blank or partially blank line, and any context after the blank is added to output.txt.

7. As stated previously, all generated questions are housed in output.txt by main.py. These can be verified within this file.

Additionally, if the user would like to run a different file, this can be done by running main.py:

```
<python or python3> main.py
```

This script will prompt the user for an input file name. As long as the file is included in the "test_files" sub-directory, it will be used to generate question objects based on the annoations in the script (explained in the next section).

Custom Syntax

For version 0.1 of the implementation, we created syntax for the input files to allow for denoting which lines of code should be in a reordering question, as well as identifying groupings of code where order does not matter. For version 0.2, these were built upon in order to add custom syntax necessary for denoting ommitted lines, grouping lines for multiple choice generation, and grouping lines for fill in the blank generation. All of these generations are now also scaleable, so multiple questions can be generated from each grouping of each type.

Grouping Questions

The code will create a grouping of these lines and mark it for the associated problem type. This annotation simply breaks the code into parts. If there is no grouping annotations, no problems will be generated.

Reordering Questions

Reordering questions are grouped by a "{*" and "*}" syntax. An example of which is shown below from the "testCodeFile.py" file in the "test_files" directory:

```
{*
# Python code for converting any number to a binary string and prints it
def convert_to_binary(num):
    if num == 0:
        return '0'
    binary = ''
    while num > 0:
        binary = str(num % 2) + binary
        num = num // 2
    return binary

# Prompt the user for a number
num = int(input('Enter a number: '))
# Convert the number to binary
binary = convert_to_binary(num)
# Print the binary string
print("The binary representation of", num, "is", binary)
*}
```

Multiple Choice Questions

Multiple choice questions are grouped by a "{$" and "$}" syntax. A single line example of which is shown below from the "testCodeFile.py" file in the "test_files" directory:

```
{$
print("The hexidecimal representation of", num, "is", hexidecimal)
$}
```

Fill in the Blank Questions

Fill in the blank questions are grouped by a "{!" and "!}" syntax. An example of which is shown below from the "testCodeFile.py" file in the "test_files" directory. NOTE: The "{@" and "@}" syntax shown within is explained in the next section:

```
{!
# Convert a binary number to hexidecimal
num = int(input('Enter a number: '))
binary = convert_to_binary(num)

{@
# Multiply the binary number by 8
binary = binary + '000'
@}

# Convert binary to hexidecimal
hexidecimal = hex(int(binary, 2))
!}
```

Additional Syntax

Lines where order does not matter

Groupings of lines where the order does not matter are grouped by a "<<<" and ">>>" syntax. An example of which from the same "testCodeFile.py" is shown below:

```
num = int(input('Enter a number: '))
binary = convert_to_binary(num)
<<<
binary = binary + '0'
binary = binary + '00'
>>>
print("The binary representation of", num, "multiplied by 8 is", binary)
```

A list of the lines between the custom syntax is saved in the reorder_question object to track these lines for the subsequent solutions.

Omitted Lines

Groupings of lines that should not be included in question generation are grouped by a "{@" and "@}" syntax. An example of which from the same "testCodeFile.py" is shown below:

```
{@
# Multiply the binary number by 8
binary = binary + '000'
@}
```

This only applies to lines within one of the aforementioned question groupings. If lines are not included in a question grouping (marked by the custom syntax), they will automatically be ignored for question generation.