PR Assignment 3: Student Record Manager

Timo Brønseth

UC1PR1102

Jan 2020

Student Record Manager

Introduction

This program presents the user with a (regrettably inconvenient) console interface for interacting with a database of students at Imaginary University. It allows the user to enter in new student records, display all student records, query the database for students taking a specific course, query the database for the oldest and youngest student(s), and, finally, it lets the user encode and decode the database using a ROT(5) Caesar shift cipher.

The program has its own class for exception handling, and all user inputs have appropriate ValueError handling. It uses no non-standard libraries, but does import string.py (access to alphabet) and textwrap.py (access to *dedent()*, to make code cleaner) for minor details.

The code has been heavily commented throughout, both with docstrings and regular comments. It's been my design philosophy that anyone browsing the code in the future should be able to understand what's going on without too much effort, even if they're fairly novice at Python.

A note on style. Both PEP8 and Google Style Guide suggests using snake_case for variable and function names, and reserve CamelCase for classes. The assignment mandated several function names with CamelCase, and lower camelCase for variables, so I decided to stay consistent with that style rather than abiding by PEP8 and my IDE's style suggestions. Consistency is key.

Implementation

The design process has been a fairly straightforward implementation of the assignment instructions, with little focus on creatively adding extras. My focus has instead been on making the code as clean and professional as I could manage, and I think I've mostly succeeded due to prioritising it heavily.

But while the design has been fairly straightforward, my choice for *how* to implement it has gone beyond what we've learned as part of the course. I use *type hinting* for some functions where I thought keeping track of types could be a challenge. Among other things, I've also used *list comprehension*, *ternary operators*, *unpacking*, *list enumeration*, *string formatting*, *context managers*, *class methods*, *static methods*, '*global' keyword*, and *recursion* for class methods. I've designed the algorithms for encrypting and decrypting from scratch. I've learned a lot from this assignment, as I'm fairly new to Python.

The program turned out well, I think, but its main weaknesses may be that I'm still uncertain about many conventions, and I haven't implemented exception handling for the files accessed. If the files get corrupted, the program crashes. I could also implement more specific ValueError handling: for example, I now check if the user enters an *integer* for student age, but I could also check whether that age is between 5 and 150.

The interface is still terrible, but it's coded to assignment specification. If I could make a simple change, I would just provide all the options in a single prompt up front, so that users don't have to wade through the annoying Y/N selections to get where they want.

Development process

I think the biggest challenges were encryption, exception handling, and learning to think about classes like a native to object-oriented programming. With regards to encryption, I found the *_SwapDigits()* function to be especially difficult for some reason.

For testing and debugging the code, I've been liberally spicing the code with print() statements everywhere. A great trick for figuring out exactly what the error is, is to use print() statements in the affected functions, and then seeing which statements get printed and which do not. That usually indicates clearly *where* the interpreter stops running, and after having localised the problem it's much easier to fix. At the end, I just tried running all the options to check if they worked as intended.

Demonstration

There are many features of this program I'd like to talk about, but I will keep it short by mentioning key parts I thought were cool.

DisplaySubjectClassList()

I thought it was cool that you could get a list of all the course-taking students with a single line of code using list comprehension:

```
# Filtering the list based on programming course using list comprehension.
subjectTakers = [student for student in studentObjects if student.programmingCourse == subjectname]
```

Exception handling

For dealing with exceptions, I decided to make my own ExceptionHandling class in a separate .py file. The class is just a collection of class methods, and it does not have an *__init__* because there is no need to create objects from it. All input queries should be called through this class, and the class does exception handling appropriate to the query.

```
class ExceptionHandlingClass:

    @classmethod
    def QueryStrGeneral(cls, queryString: str, errorPrompt: str, conditionList: list) -> str:
        """A more general version, with more arguments, to query user for a string."""

        # userInput needs to be raised to global so that it can be updated
        # while recursively calling this function in the except clause.
        global userInput

        try:
            userInput = input(queryString).upper()

            # Check if userInput points to either of the options,
            # and raise exception if it does not.
            if userInput not in conditionList:
                raise ValueError

        except ValueError:
            # If ValueError, recursively call the function until the user enters an actionable value.
            print(errorPrompt)
            cls.QueryStr(queryString, errorPrompt, conditionList)

        except Exception:
            # Only runs for errors which are not ValueErrors.
            # Assignment says to catch all input errors like this.
            print("\nOops something is buggy")

        return userInput
```

An example of a class method defined inside ExceptionHandlingClass.

Note the *cls.QueryStr(...)* call in the except clause: that's recursion.

And for an example of calling the method:

```
def DisplayOptions():
    """Provides the user the option of using several accessor methods to display information
    from the database."""

    # Ask for user input via the ErrorHandling method.
    # textwrap.dedent() removes the indentations from the string.
    queryString = dedent("""
                            1. Would you like to see a list of all registered students?
                            2. Would you like to see a class list for a specific subject?
                            3. Would you like to see who your oldest student is?
                            4. Would you like to see who your youngest student is?
                            Enter a number for the selected task, or X to skip this: """)
    errorPrompt = "\nPlease select either 1, 2, 3, 4 or X."
    conditionList = ["1", "2", "3", "4", "X"]
    userInput = ExceptionHandling.QueryStrGeneral(queryString, errorPrompt, conditionList)
```

An example of the QueryStrGeneral() method being called.

The *QueryStrGeneral()* method takes three arguments:

- *queryString* is the string that prompt the user to enter an input.
- *errorPrompt* is the string that gets printed to the user if they entered a wrong input.
- *conditionList* is a list of options that the user can enter, otherwise a ValueError is raised.

Encryption

This is the encryption algorithm I thought was pretty cool. Docstring shortened to make room for the screenshot, but read the comments and you should get a good idea of what's going on. Also, there's *type hinting* in the definition, indicating what types each argument should take, and what type the function returns.

```
@classmethod
def CaesarShiftPlus(cls, plainText: str, numShifts: int) -> str:
    """..."""

    # cipherText starts out as a plaintext list, and then gets encrypted.
    cipherText = list(plainText)  # Converts string to a list of characters.

    # Implement the Caesar shift.
    for i, character in enumerate(plainText):
        character = character.lower()  # Convert to lower case.
        if character in cls._ALPHABET:
            indexInAlphabet = cls._ALPHABET.find(character)
            cipherText[i] = cls._ALPHABET[(indexInAlphabet + numShifts) % 26]

    # Converts cipherText from list of characters to a string.
    cipherText = "".join(cipherText)

    # Applies the SwapDigits cipher, swapping the
    # first and last digit for each number in the text.
    cipherText = _SwapDigits(cipherText)

    # Capitalizes names in the text.
    cipherText = _CapitalizeNames(cipherText)

    return cipherText
```