August Belhumeur
11/19/2023
IT FDN 110
Assignment 06 - Document Knowledge
Github Link

Functions and Classes

Introduction

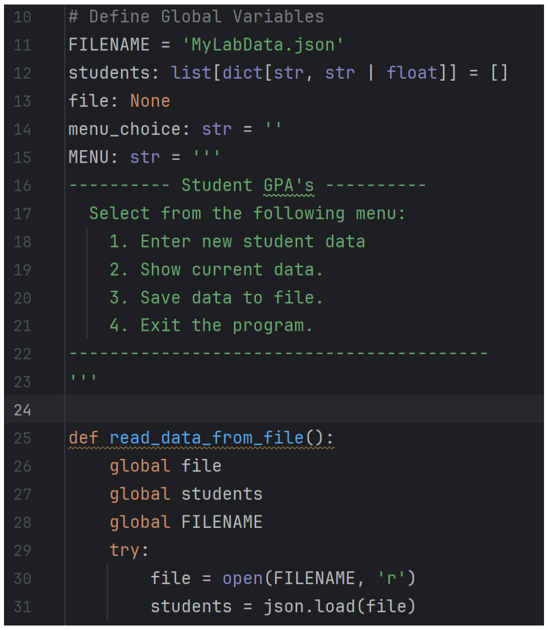
Module 06 focuses on functions and classes and how both can be used to organize repeated code into structures with more modular, autonomous, and reusable entities. The module begins with lessons on functions, which are blocks of stand alone code that run only when called. We learned about the different parts of a function, including parameters, arguments, and return values, so that we know how to properly format a function and input and return values correctly. Then, the module expands on these lessons to instruct on how to use classes to organize functions so that related methods can be bundled together and reused, even across multiple projects. These methods allow us to better organize code for clarity and separation. At the end of the module, we re-structure Assignment 05's program for Assignment 06 to showcase classes, functions, arguments, and returns. The resulting code needs to have more clarity in both documentation and structure.

Functions

A function is a collection of actions that gets grouped together under a single abstracted and reusable block. They're useful when there is duplicated logic that gets called again and again in a program that could be replaced by a single line of code calling a function instead of duplicating (and decoupling) the same code over again. Every function needs to be defined in a **"def"** statement, where you define the name of the function and put all of the code that you want it to call underneath. Python reads code from top to bottom, so functions should always be ordered with this in mind.

All variables in a program are **local** or **global**. Global variables are established at the start of the program where constants and variables are defined. They're active parameters that are always available for use and are the main type of variable that we've worked with thus far. Any variable inside of a function not specifically set to correspond with a global variable will be local, which means that it's only usable inside of the function and ceases to exist once you leave the function. The code block of any function that contains the names that you define inside of the function is called **scoping** and these names are only visible from the code of the function. **(See Figure A)** Additionally, if you wanted to merge a local variable into a global one, you could use either a "+=" operator or a for loop to add for an individual entry in your local list and append to the global variable.

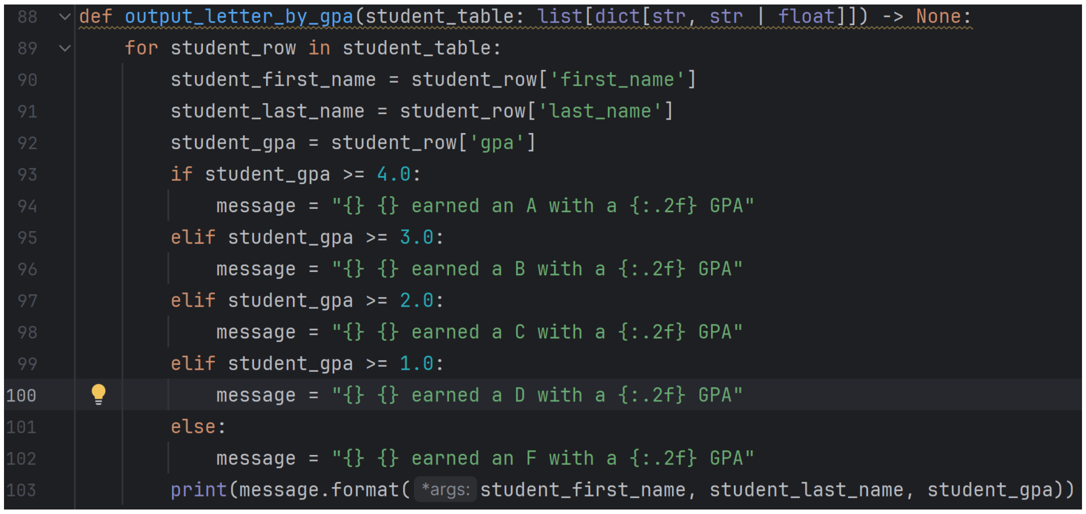
August Belhumeur
11/19/2023
IT FDN 110
Assignment 06 - Document Knowledge
Github Link



```python
# Define Global Variables
FILENAME = 'MyLabData.json'
students: list[dict[str, str | float]] = []
file: None
menu_choice: str = ''
MENU: str = '''
---------- Student GPA's ----------
  Select from the following menu:
    1. Enter new student data
    2. Show current data.
    3. Save data to file.
    4. Exit the program.
-----------------------------------------
'''

def read_data_from_file():
    global file
    global students
    global FILENAME
    try:
        file = open(FILENAME, 'r')
        students = json.load(file)
```

Figure A: Example code of defining global variables within a function.

A **parameter** is a type of data that gets passed through a function through the parenthesis at the end of the "def" (definition) line of code. They can take the place of global variables and reduce the use of global variables in functions. Using global variables in functions can get messy if the function gets used across multiple files. Most global variables could be a parameter with defined data types instead. Then, when the function is called elsewhere in the program, new variables can be passed through the argument, but they need to line up with the parameters defined in the function to pass different data through the function. **(See Figure B)**

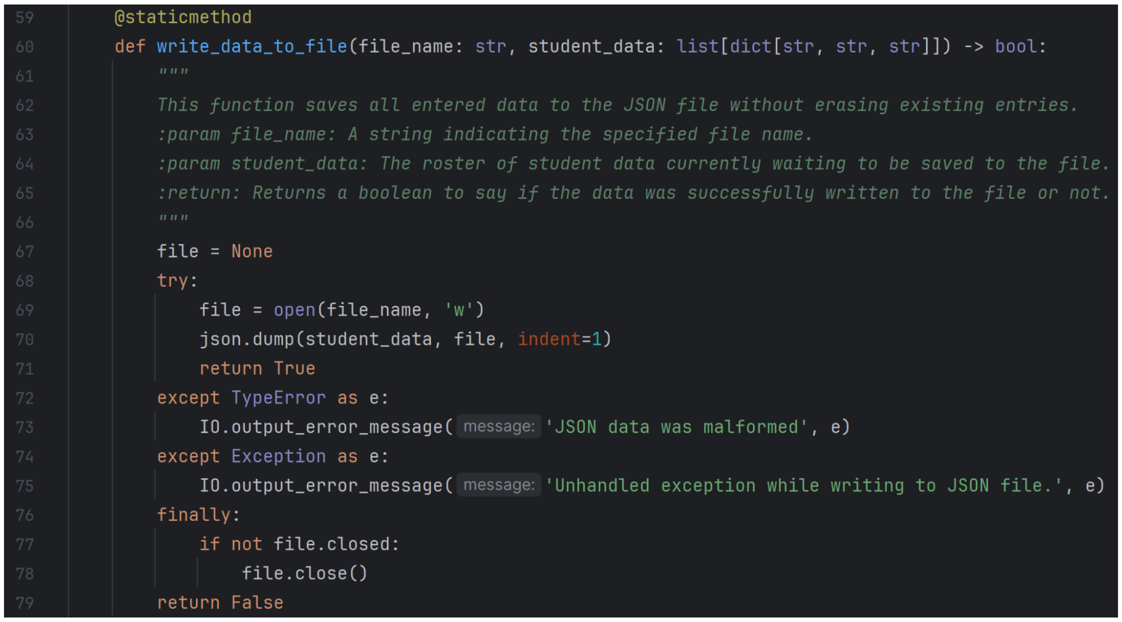


```python
def output_letter_by_gpa(student_table: list[dict[str, str | float]]) -> None:
    for student_row in student_table:
        student_first_name = student_row['first_name']
        student_last_name = student_row['last_name']
        student_gpa = student_row['gpa']
        if student_gpa >= 4.0:
            message = "{} {} earned an A with a {:.2f} GPA"
        elif student_gpa >= 3.0:
            message = "{} {} earned a B with a {:.2f} GPA"
        elif student_gpa >= 2.0:
            message = "{} {} earned a C with a {:.2f} GPA"
        elif student_gpa >= 1.0:
            message = "{} {} earned a D with a {:.2f} GPA"
        else:
            message = "{} {} earned an F with a {:.2f} GPA"
        print(message.format( *args: student_first_name, student_last_name, student_gpa))
```

Figure B: Example code using parameters to define data types that passes new variables through the argument when the function is called elsewhere in the program.

August Belhumeur
11/19/2023
IT FDN 110
Assignment 06 - Document Knowledge
Github Link

At the end of a function, you can return data by using the defined parameters instead of mutating the global parameter. Python automatically returns “None” from a function that isn’t writing or returning anything from the code, so you can keep the code cleaner by marking space for the program to have the potential to return something by manually adding in a return that has the program return “None”. Functions can also return booleans, which can be “success” functions. For example, if you’re writing a try/except statement, the code will “try” to run the block of code and return true if the code is successful. However, if it hits any of the exception errors written below, then it returns false, meaning that it didn’t succeed at running the initial “try” code block. The function stops executing when it hits a return statement. **(See Figure C)** Otherwise, return statements can be used to output the results of code in a function, which could be any kind of value in a variable that can then be used outside of the function or saved in a different temporary variable.

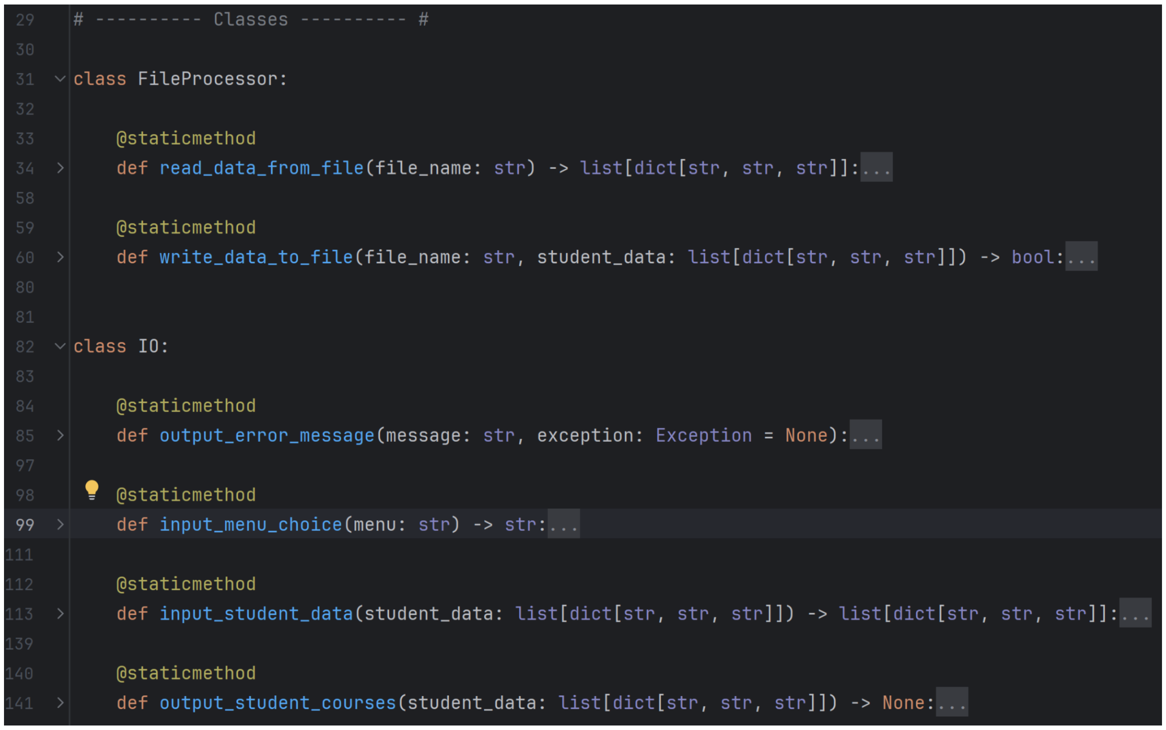


```
    @staticmethod
    def write_data_to_file(file_name: str, student_data: list[dict[str, str, str]]) -> bool:
        """
        This function saves all entered data to the JSON file without erasing existing entries.
        :param file_name: A string indicating the specified file name.
        :param student_data: The roster of student data currently waiting to be saved to the file.
        :return: Returns a boolean to say if the data was successfully written to the file or not.
        """
        file = None
        try:
            file = open(file_name, 'w')
            json.dump(student_data, file, indent=1)
            return True
        except TypeError as e:
            IO.output_error_message( message: 'JSON data was malformed', e)
        except Exception as e:
            IO.output_error_message( message: 'Unhandled exception while writing to JSON file.', e)
        finally:
            if not file.closed:
                file.close()
        return False
```

Figure C: Example code of a function returning “true” or “false” in a try/except statement to determine if the code block under “try” was able to complete successfully.

Classes

A **class** is a group of related functions grouped under a single object and given a name that explains the overall use of the group. For example, if you have a class that contains all functions regarding the use of external files, you could call it “class FileProcessor”. This allows you to create your own data structures containing related code. Functions used inside of classes are called **methods** and need to be defined with the line of code “@staticmethod” prior to writing the function and its associated code block. They’re used inside classes to perform operations on the type of data described. Classes can call multiple functions with related attributes and methods and can even call other classes and use them in the nested functions, but they need to be defined in the order they're used from top to bottom. This helps to better organize code using modular and autonomous entities that have reuse value in multiple projects because they're standalone. **(See Figure D)** Lastly, separating classes into these distinct pieces of logic is called "Separations of Concern". It keeps sections of logic from getting too intertwined. The goal of such code is clarity and separation. Reusability is considered a secondary benefit.



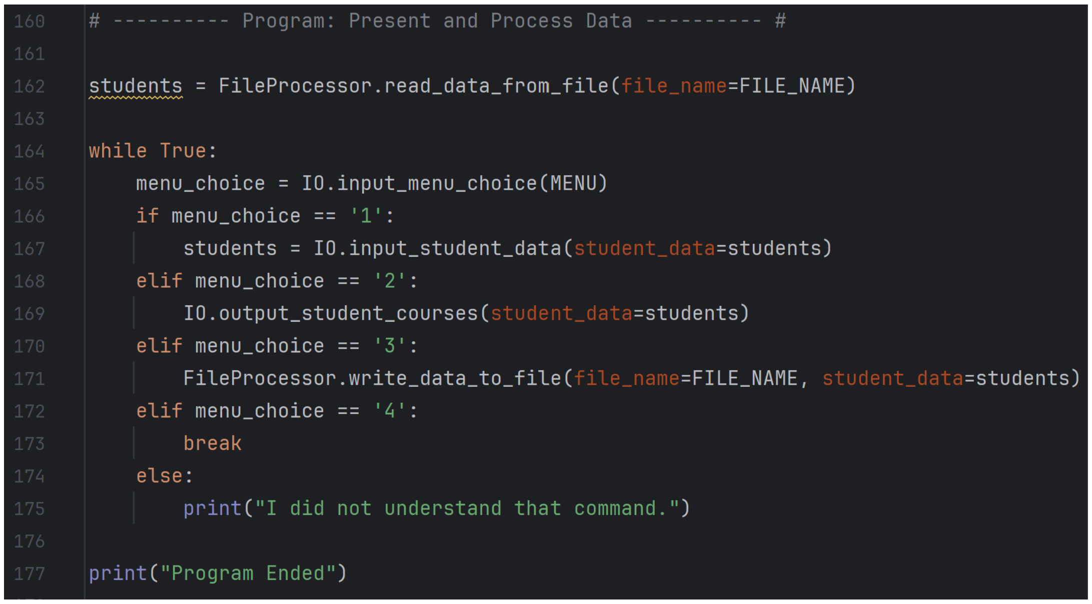
```
# ---------- Classes ---------- #

class FileProcessor:

    @staticmethod
    def read_data_from_file(file_name: str) -> list[dict[str, str, str]]:...

    @staticmethod
    def write_data_to_file(file_name: str, student_data: list[dict[str, str, str]]) -> bool:...


class IO:

    @staticmethod
    def output_error_message(message: str, exception: Exception = None):...

    @staticmethod
    def input_menu_choice(menu: str) -> str:...

    @staticmethod
    def input_student_data(student_data: list[dict[str, str, str]]) -> list[dict[str, str, str]]:...

    @staticmethod
    def output_student_courses(student_data: list[dict[str, str, str]]) -> None:...
```

Figure D: Example code for how to structure classes with multiple nested functions.

Creating the Script for Assignment 06

The goal of this assignment is not to drastically change the result of the program from Assignment 05, but to change the structure of the code to group it in reusable functions and group related functions into classes. This makes the program more organized, with better clarity and less duplicated / decoupled code. I started by changing the program's header, data constants, and data variables. While the constants remain largely the same, the data variables are significantly reduced to only require two global variables "students" and "menu_choice" as all other variables become local variables in functions. Next, I outlined the core structure of the program under the comment header "Program: Present and Process Data" to be able to visualize the overall order that the program should run in. Under each menu choice, I defined the name of the appropriate function to call for that selection, as outlined in the instructions

August Belhumeur
11/19/2023
IT FDN 110
Assignment 06 - Document Knowledge
Github Link
document. In the program, this section of code exists at the end of the script, but I wrote it first as advised in the demos to ensure I knew what functions I want to use in the program and how they should be organized by class. **(See Figure E)**

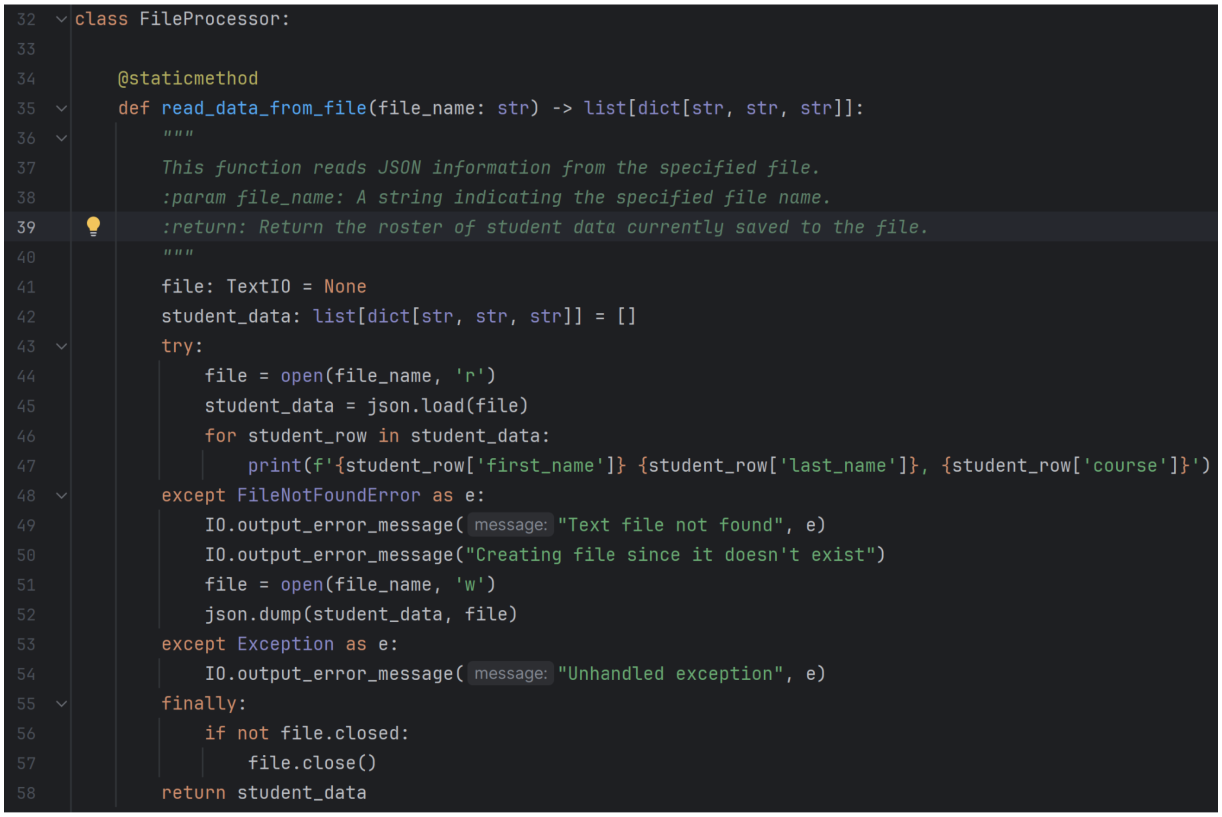


```
# ---------- Program: Present and Process Data ---------- #

students = FileProcessor.read_data_from_file(file_name=FILE_NAME)

while True:
    menu_choice = IO.input_menu_choice(MENU)
    if menu_choice == '1':
        students = IO.input_student_data(student_data=students)
    elif menu_choice == '2':
        IO.output_student_courses(student_data=students)
    elif menu_choice == '3':
        FileProcessor.write_data_to_file(file_name=FILE_NAME, student_data=students)
    elif menu_choice == '4':
        break
    else:
        print("I did not understand that command.")

print("Program Ended")
```

Figure E: Assignment 06's code for presenting and processing data.

The first class is called "FileProcessor" and contains a function for reading data from a file and a function for writing data to a file. On start, the program calls the "read_data_from_file" function to print all JSON data from the specified file to the user at the start of the program. The function includes documentation to explain the use of the function, what the parameters do, and what the function will return, which is the "student_data" roster (a list of dictionaries) that is loaded from the file. The dictionaries in this list get printed to the user. The function has try/except error handling for dealing with a FileNotFound error and any unhandled exceptions. **(See Figure F)**

August Belhumeur
11/19/2023
IT FDN 110
Assignment 06 - Document Knowledge
Github Link



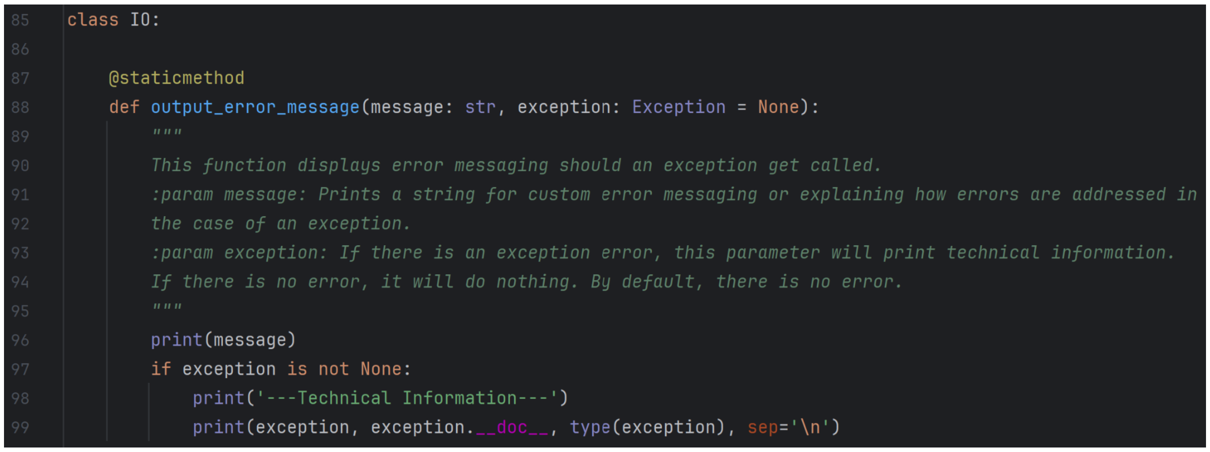
```python
class FileProcessor:

    @staticmethod
    def read_data_from_file(file_name: str) -> list[dict[str, str, str]]:
        """
        This function reads JSON information from the specified file.
        :param file_name: A string indicating the specified file name.
        :return: Return the roster of student data currently saved to the file.
        """
        file: TextIO = None
        student_data: list[dict[str, str, str]] = []
        try:
            file = open(file_name, 'r')
            student_data = json.load(file)
            for student_row in student_data:
                print(f'{student_row['first_name']} {student_row['last_name']}, {student_row['course']}')
        except FileNotFoundError as e:
            IO.output_error_message( message: "Text file not found", e)
            IO.output_error_message("Creating file since it doesn't exist")
            file = open(file_name, 'w')
            json.dump(student_data, file)
        except Exception as e:
            IO.output_error_message( message: "Unhandled exception", e)
        finally:
            if not file.closed:
                file.close()
        return student_data
```

Figure F: A function from Assignment 06 under the class "FileProcessor" called "read_data_from_file".

The function "write_data_to_file" in the class "FileProcessor" gets called in menu option 3 where the user can save all input student registrations to the same JSON file. Just like the previous function, this function includes documentation for what it does, the parameters, and what the function will return, which is a success boolean to say if the data was successfully written to the file. It also includes error handling for TypeErrors or any unhandled exceptions. **(See Figure C)** Those are all of the functions under the class "FileProcessor".

The second class is called "IO", which stands for "Input / Output". This class handles all related to data being input by the user or output to the user. The first function in this class is "output_error_message" which groups error handling code into a single line of code when called by a function so that the full code block doesn't need to be duplicated for each instance of error handling in other functions. An example of this function being called in another function can be seen in the previous **Figure F** wherever it says "IO.output_error_message"**.** This function prints out any custom messaging for the specific error as well all related technical information from Python. **(See Figure G)**

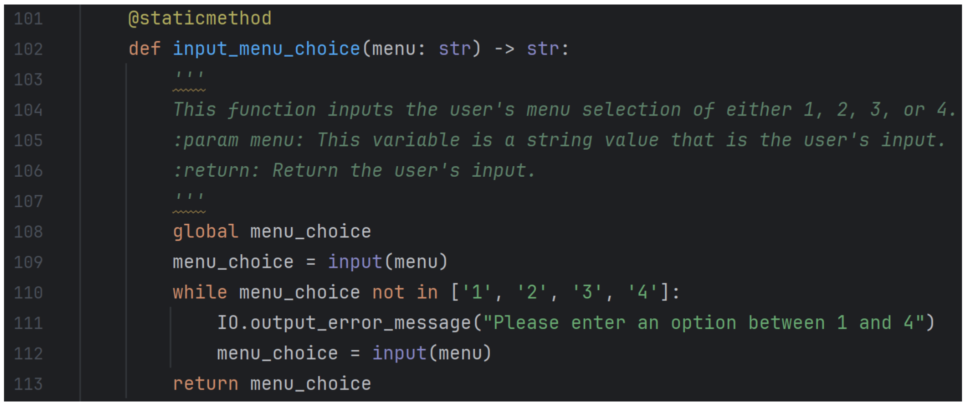
August Belhumeur
11/19/2023
IT FDN 110
Assignment 06 - Document Knowledge
Github Link



```python
class IO:

    @staticmethod
    def output_error_message(message: str, exception: Exception = None):
        """
        This function displays error messaging should an exception get called.
        :param message: Prints a string for custom error messaging or explaining how errors are addressed in
        the case of an exception.
        :param exception: If there is an exception error, this parameter will print technical information.
        If there is no error, it will do nothing. By default, there is no error.
        """
        print(message)
        if exception is not None:
            print('---Technical Information---')
            print(exception, exception.__doc__, type(exception), sep='\n')
```

Figure G: A function from Assignment 06 called "output_error_message".

Next, I added the function for "input_menu_choice" which gets called in the program's while loop for the user to input which menu action they'd like to perform. The function inputs the user's menu selection and handles a user input error should the user try to input anything other than 1, 2, 3, or 4. The function includes documentation for what the function does, what the listed parameter is, and what the function returns, which is the user's input. **(See Figure H)**

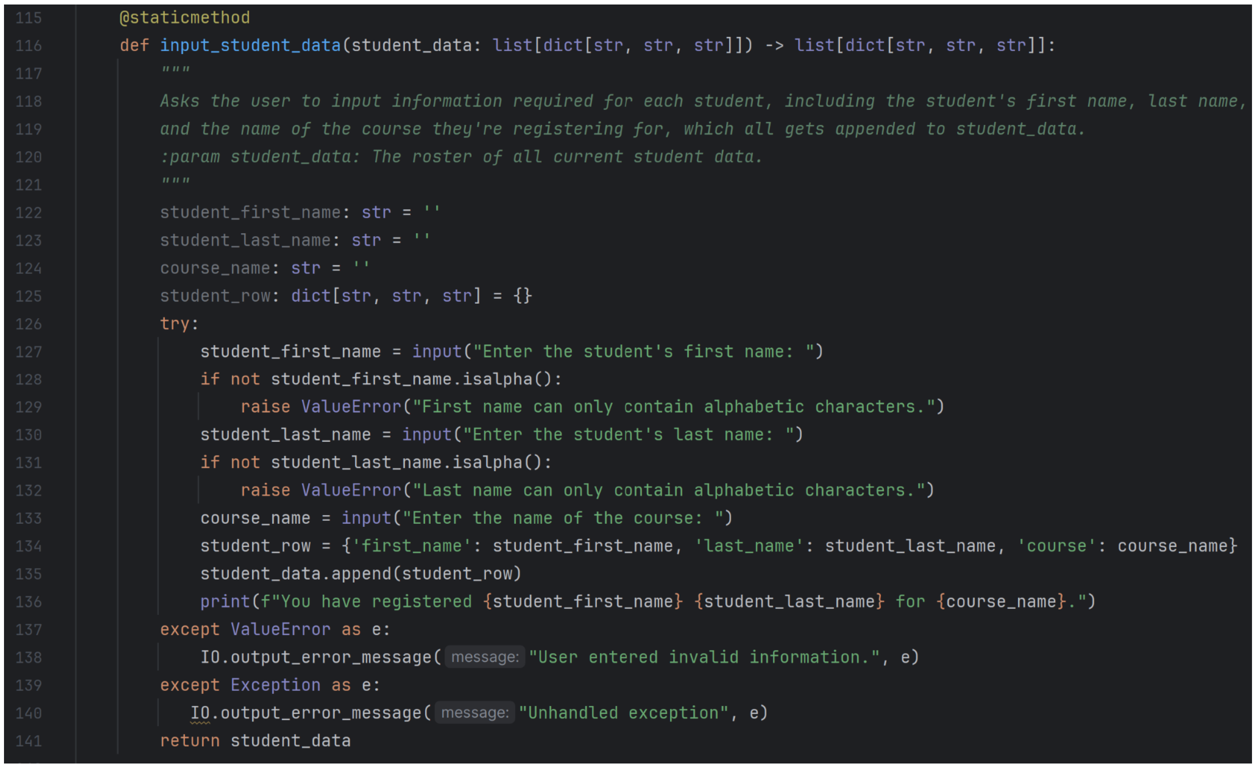


```python
    @staticmethod
    def input_menu_choice(menu: str) -> str:
        '''
        This function inputs the user's menu selection of either 1, 2, 3, or 4.
        :param menu: This variable is a string value that is the user's input.
        :return: Return the user's input.
        '''
        global menu_choice
        menu_choice = input(menu)
        while menu_choice not in ['1', '2', '3', '4']:
            IO.output_error_message("Please enter an option between 1 and 4")
            menu_choice = input(menu)
        return menu_choice
```

Figure H: A function from Assignment 06 called "input_menu_choice".

Next, I added the function for "input_student_data" to class "IO" which contains all of the code necessary to gather the user's input for student registration, including the name of the student and the name of the course they're registering for. This function gets called under menu choice 2 and includes error handling to ensure that only alphabetical characters can be input on student names. It appends all entered data into the list of dictionaries "student_data" and returns that variable. **(See Figure I)**

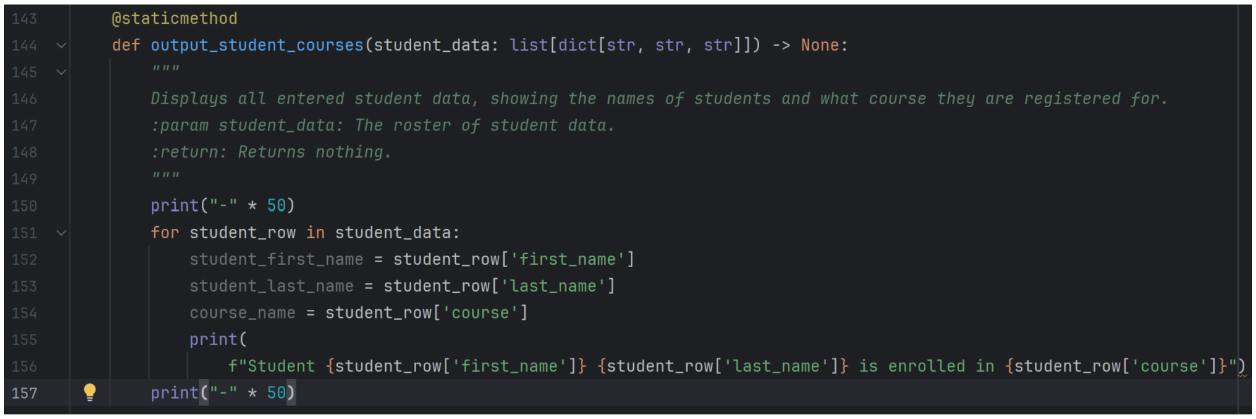
August Belhumeur
11/19/2023
IT FDN 110
Assignment 06 - Document Knowledge
[Github Link]



```python
    @staticmethod
    def input_student_data(student_data: list[dict[str, str, str]]) -> list[dict[str, str, str]]:
        """
        Asks the user to input information required for each student, including the student's first name, last name,
        and the name of the course they're registering for, which all gets appended to student_data.
        :param student_data: The roster of all current student data.
        """
        student_first_name: str = ''
        student_last_name: str = ''
        course_name: str = ''
        student_row: dict[str, str, str] = {}
        try:
            student_first_name = input("Enter the student's first name: ")
            if not student_first_name.isalpha():
                raise ValueError("First name can only contain alphabetic characters.")
            student_last_name = input("Enter the student's last name: ")
            if not student_last_name.isalpha():
                raise ValueError("Last name can only contain alphabetic characters.")
            course_name = input("Enter the name of the course: ")
            student_row = {'first_name': student_first_name, 'last_name': student_last_name, 'course': course_name}
            student_data.append(student_row)
            print(f"You have registered {student_first_name} {student_last_name} for {course_name}.")
        except ValueError as e:
            IO.output_error_message( message: "User entered invalid information.", e)
        except Exception as e:
            IO.output_error_message( message: "Unhandled exception", e)
        return student_data
```

Figure I: A function from Assignment 06 called "input_student_data".

The last function under class "IO" is "output_student_courses", which gets called in menu choice 3 and displays all entered student data back to the user. It uses a for statement to print each "student_row" dictionary entry within the list "student_data". It returns nothing after the function has finished printing the data. The function includes documentation to explain the actions of the function, the parameters used (in this case, "student_data"), and what the function returns, which is "None". **(See Figure J)**



```python
    @staticmethod
    def output_student_courses(student_data: list[dict[str, str, str]]) -> None:
        """
        Displays all entered student data, showing the names of students and what course they are registered for.
        :param student_data: The roster of student data.
        :return: Returns nothing.
        """
        print("-" * 50)
        for student_row in student_data:
            student_first_name = student_row['first_name']
            student_last_name = student_row['last_name']
            course_name = student_row['course']
            print(
                f"Student {student_row['first_name']} {student_row['last_name']} is enrolled in {student_row['course']}")
        print("-" * 50)
```

Figure J: A function from Assignment 06 called "output_student_courses".

August Belhumeur
11/19/2023
IT FDN 110
Assignment 06 - Document Knowledge
Github Link

Summary

In this module, we learned about functions and classes and how to organize these structures of repeated code into modular and reusable entities. We learned about how to group code into functions, formatting variables for global or local use, and how to specify parameters and arguments for the new functions. We learned how to organize functions into classes so that related methods can be grouped together and reused across a program or even multiple programs. We also began adding documentation to blocks of code, explaining what a given function does, what the attached parameters are for, and what the function returns, if anything. This allows us to write code that is more organized, separated, and clear. At the end of the module, we re-formatted Assignment 05 so that all actions in the program are grouped into functions and then further organized into classes. The resulting code functions almost identically to Assignment 05, but it's more clear in both documentation and structure.