Nathan Parece

5/28/2025

IT FDN 110 A

Assignment 06

https://github.com/Nathan-Parece/IntroToProg-Python-Mod06

Classes and Functions

Introduction

This week presented a major change to the structure of our student registration program. While the basic function of the program remains the same, we shrunk the main body down to just sixteen lines of code, not including comments and whitespace. We accomplished this using classes and functions. These tools let us break up our code into small pieces that can be organized, reused, and updated independently from one another, extending Python's library of built-in functions to meet the exact needs of our program.

Header, Modules, Constants, Variables

At the top of our program, we have the header, modules, constants, and variables. The header is the same, and we're not using any new modules or constants.

```
# ------------------------------------------------------
# Title: Assignment06_Starter
# Desc: This assignment demonstrates using functions
# with structured error handling
# Change Log: (Who, When, What)
#   NParece, 5/28/2025, Created Script
# ------------------------------------------------------

import json
```

Fig. 1: Header and Modules

```
# Constants
MENU: str = '''
---- Course Registration Program ----
  Select from the following menu:
    1. Register a Student for a Course
    2. Show current data
    3. Save data to a file
    4. Exit the program
----------------------------------------
'''
FILE_NAME: str = "Enrollments.json"
```

Fig. 2: Constants

But the variables look very different from how they did last week. Now, we only have two: students and menu_choice. These are global variables, which means that they exist outside of the classes and functions defined later in the code. The students table is global because it's used and modified by multiple different functions in our program, and its value needs to persist between loops. The menu_choice string is global because it needs to exist outside of any function so it can be referenced by the main loop.

```
# Variables
students: list = []  # A table of student data
menu_choice: str = ''  # Holds the choice made by the user
```

Fig. 3: Variables

All of the other variables are no longer global! They've been encapsulated into functions so that they only get referenced when they're supposed to. This prevents side effects and makes our code easier to maintain, but for now, it just means we don't have to define them at the top.

The FileProcessor Class

Our program has two classes, or groups of functions. Organizing functions into classes keeps similar functions close together and helps us impose limits on scope. The FileProcessor class is for the functions that interact with text files. It opens with a docstring describing what the class is and what's in it.

```
class FileProcessor:  2 usages
    """
    A collection of functions that read and write JSON data to files

    ChangeLog: (Who, When, What)
    NParece,5.28.2025,Created Class
    NParece,5.28.2025,Added methods to read and write data
    """
```

Fig. 4: The FileProcessor Class

There's some small text that says "2 usages" next to the first line of the class. This is a tip from PyCharm, letting us know that the FileProcess class is used twice in the main body of the code.

Inside the FileProcessor class are two functions: read_data_from_file and write_data_to_file. Each of these takes a function that was formerly contained in the main body of this program and encapsulates it so that it can be called anywhere with just one line. First, let's look at the read_data_from_file function.

read_data_from_file

This function opens a json file and pulls the data into the program. It also has some error handling. We start out by using the @staticmethod decorator, which tells Python that the code in this function isn't going to be modified while the script is running (it's static). We also include a docstring explaining what the function is for, what each of the parameters passed to the function should be, and what the function returns. All together, it looks like this:

```
@staticmethod  1 usage
def read_data_from_file(file_name: str, student_data: list):
    """ This function reads JSON data from a file into a local variable

    ChangeLog: (Who, When, What)
    NParece,5.28.2025,Created function

    :param file_name: Name of the file
    :param student_data: List of dicts containing student data
        (empty on initialization)

    :return: JSON data from referenced file
    """
```

Fig. 5: read_data_from_file Definition

Docstrings are an important part of good code. In practice, they are intended to make things like this knowledge document obsolete by letting the code itself explain what it is supposed to do.

This function has two parameters. Parameters act like local variables, with the added benefit that they can be assigned values when the function is called. In fact, we've been using parameters this whole time–every time the print() function is called with a string inside the

parenthesis, that string is filling in one of the print() function's parameters. Our function needs a file name to reference and a list to fill with data, so we include parameters to pass these in when we use the function.

The structure of the function is nearly identical to the segment of code before the main loop in last week's version of this program. We use a try-except block to open and read a file, and handle any errors using custom error messages.

```
try:
    file = open(file_name, "r")
    student_data = json.load(file)
except FileNotFoundError as e:
    IO.output_error_messages(
        message: "Text file must exist before running this script!", e)
except Exception as e:
    IO.output_error_messages(
        message: "There was an unspecified error!", e)
```

Fig. 6: read_data_from_file Try Block

Note that we're using the output_error_messages function from the IO class to actually display our custom error messages. This will be covered later. We're also no longer updating the students global variable in place–instead, we're using the local variable student_data.

The finally block of the try-except block has also been changed. Previously, we ended the try-except block with an if statement that would close the file if it failed to close earlier. Unfortunately, this would cause an unhandled error if the file did not exist, because the open() method would fail, the file local variable would never get defined, and the .closed method can't act on a variable that doesn't exist. This error was very tricky to handle! If the file variable doesn't exist, we can't use it in an if statement, and we can't define it arbitrarily because we might need to close the file object. I decided to use a nested try-except block:

```
finally:
    try:
        file.close()
    except UnboundLocalError as e:
        IO.output_error_messages(
            message: "You can't close a file that doesn't exist!", e)
    except Exception as e:
        IO.output_error_messages(
            message: "There was an unspecified error!", e)
```

Fig. 7: read_data_from_file Finally Block

Because the finally block always runs, we no longer need to include file.close() in the initial try block. By including it here instead, we capture both the FileNotFoundError and the UnboundLocalError when there's no Enrollments.json file, and we still close the file during normal operation.

Last, we return the student_data variable, which lets us set the students global variable later on in the main body.

```
return student_data
```

Fig. 8: read_data_from_file Return

write_data_to_file

This function does what menu option 3 of the main loop did in last week's version. We include the @staticmethod decorator and a docstring.

```
@staticmethod  1 usage
def write_data_to_file(file_name: str, student_data: list):
    """ This function writes JSON data from a variable into a text file

    ChangeLog: (Who, When, What)
    NParece,5.28.2025,Created function

    :param file_name: Name of the file
    :param student_data: List of dicts containing student data
        (defined during operation)

    :return: None
    """
```

Fig. 9: write_data_to_file Definition

Like the previous function, this function uses a file name and the list of student data. The docstring is different, however–we expect to be using this function after the students variable has some data in it, so we include that information in the docstring.

The main logic of this function is what we expect, and we've placed it in a try-except block for error handling. We include a loop to print the data that was saved; while this could be an IO function, it wasn't included in the assignment, so we keep it where it was.

```
try:
    file = open(file_name, "w")
    json.dump(student_data, file)
    print("The following data was saved to file!")
    for student in student_data:
        print(f'Student {student["FirstName"]} '
              f'{student["LastName"]} is enrolled in {student["CourseName"]}')
except TypeError as e:
    IO.output_error_messages(
        message: "Please ensure the data is in JSON format.", e)
except Exception as e:
    IO.output_error_messages(
        message: "There was a problem with writing to the file.", e)
```

Fig. 10: write_data_to_file Try Block

As with the previous function, we've also updated finally block to handle closing the file and catching any weird errors.

```
finally:
    try:
        file.close()
    except UnboundLocalError as e:
        IO.output_error_messages(
            message: "You can't close a file that doesn't exist!", e)
    except Exception as e:
        IO.output_error_messages(
            message: "There was an unspecified error!", e)
```

Fig. 11: write_data_to_file Finally Block

This function doesn't return anything–it just copies data from the program to a file.

The IO Class

Next up, we have the IO class. This class is for interacting with a user. We open the class with a docstring.

```
class IO:
    """
    A collection of functions that handle input and output of student
    registration data, as well as navigation of an external text menu

    ChangeLog: (Who, When, What)
    NParece,5.28.2025,Created Class
    NParece,5.28.2025,Added methods to output the menu, errors,
    and registration data
    NParece,5.28.2025,Added methods to input menu choice and registration data
    """
```

Fig. 12: The IO Class

This class has five functions. Each captures a basic function of last week's main body.

output_error_messages

This function is the most reused function in the program. It is responsible for formatting the strings that tell a user when an error happens. It also contains the only instance of a default value in this program, by setting the default value of the error parameter to None. This lets us opt not to display the technical error message. Here's the decorator, definition, and docstring:

```
@staticmethod  11 usages
def output_error_messages(message: str, error: Exception = None):
    """ This function outputs formatted error messages to the console

    ChangeLog: (Who, When, What)
    NParece,5.28.2025,Created function

    :param message: Custom error message
    :param error: Python error data, defaults to None

    :return: None
    """
```

Fig. 13: output_error_messages Definition

And the functional code:

```
print(message, end = "\n\n")

if error is not None:
    print("-- Technical Error Message -- ")
    print(error, error.__doc__, type(error), sep = "\n")
```

Fig. 14: output_error_messages Logic

This structure lets each error handling block define its own custom error message and display lots of information in just one line. Naturally, this function doesn't return anything.

output_menu

This function outputs the MENU constant. It's called every time the main loop runs. Here it is, in its entirety:

```
@staticmethod  1 usage
def output_menu(menu: str):
    """ This function prints a string to the console

    ChangeLog: (Who, When, What)
    NParece,5.28.2025,Created function

    :param menu: The string to be printed.
        This should always be the MENU global constant

    :return: None
    """
    print(menu)
```

Fig. 15: output_menu

There are lots of ways to accomplish this with fewer lines of code, but this method has the advantage of clarity. When it shows up in the main loop, we know exactly what it does. We could also have a variety of menus, because this function doesn't care what it's passed; this could be useful in more complicated programs with sub-menus.

input_menu_choice

This function handles the user's input of a menu option. It uses a local variable called choice to update the global menu_choice variable. It also has some error handling in case a user tries to input an invalid menu option. Here's the decorator, definition, and docstring, as well as the definition of the local variable:

```
@staticmethod  1 usage
def input_menu_choice():
    """ This function handles user input to choose a menu option

    ChangeLog: (Who, When, What)
    NParece,5.28.2025,Created function

    :return: The menu choice string
    """
    choice: str = "0"
```

Fig. 16: input_menu_choice Definition

And here's the try-except block:

```
try:
    choice = input("What would you like to do: ")
    if choice not in ("1", "2", "3", "4"):
        raise Exception("Please choose a valid option!")
except Exception as e:
    IO.output_error_messages(e.__str__())
return choice
```

Fig. 17: input_menu_choice Logic

Note that while we pass the custom error message to output_error_messages, we don't actually pass an error. This causes the if statement in that function to get skipped, meaning the user will only see our custom error message.

output_student_courses

This function replaces menu option 2. Here's the decorator, definition, and docstring:

```
@staticmethod  1 usage
def output_student_courses(student_data: list):
    """ This function outputs all student registrations in csv format

    ChangeLog: (Who, When, What)
    NParece,5.28.2025,Created function

    :param student_data: List of dicts containing student data,
        including new entries

    :return: None
    """
```

Fig. 18: output_student_courses Definition

And here's the code:

```
print("-" * 50)
for student in student_data:
    print(f'{student["FirstName"]}, '
          f'{student["LastName"]}, {student["CourseName"]}')
print("-" * 50)
```

Fig. 19: output_student_courses Logic

This function is nice and simple. When we pass it the students list of dicts, it loops over each of the contained dicts and displays the data in csv format. It doesn't return anything.

input_student_data

This function replaces menu option 1. We've replaced the error handling code with calls to output_error_messages, but otherwise it is nearly unchanged. We do need to pass in the students variable so that we can append the new data, but as with all other usages of this variable, we pass it into a parameter rather than using a global variable. Here's the decorator, definition, and docstring:

```
@staticmethod  1 usage
def input_student_data(student_data: list):
    """ This function allows a user to input one new student registration

    ChangeLog: (Who, When, What)
    NParece,5.28.2025,Created function

    :param student_data: List of dicts containing student data,
        including new entries

    :return: Updated list of student data
    """
```

Fig. 20: input_student_data Definition

And the try-except block, followed by a return statement to get the new data back into the students global variable:

```
try:
    student_first_name: str = input("Enter the student's first name: ")
    if not student_first_name.isalpha():
        raise ValueError("The first name should contain only letters.")
    student_last_name: str = input("Enter the student's last name: ")
    if not student_last_name.isalpha():
        raise ValueError("The last name should contain only letters.")
    course_name: str = input("Please enter the name of the course: ")
    student: dict = {"FirstName": student_first_name,
                     "LastName": student_last_name,
                     "CourseName": course_name}
    student_data.append(student)
    print(f"You have registered {student_first_name} {student_last_name} "
          f"for {course_name}.")
except ValueError as e:
    IO.output_error_messages(e.__str__(), e)
except Exception as e:
    IO.output_error_messages(
        message: "There was a problem with your entered data.", e)
return student_data
```

Fig. 21: input_student_data Logic

This is the last function we need, so we can now write the main body.

The Main Body

Now that everything the code needs to do is contained in functions, the main body is very short. We start by extracting any pre-existing data from the Enrollments.json file:

```
# When the program starts, read the file data into a list of dicts (table)
# Extract the data from the file
students = FileProcessor.read_data_from_file(FILE_NAME, students)
```

Fig. 22: Read the File

Next, we start the main loop, present the menu, and get the user's menu choice:

```
# Main Loop
while True:

    # Present the menu of choices
    IO.output_menu(MENU)
    menu_choice = IO.input_menu_choice()
```

Fig. 23: Loop, Display Menu, Get Menu Choice

Then, we start our if-else block. If the user inputs 1, we'll register a new student:

```
# Input user data
if menu_choice == "1":

    students = IO.input_student_data(students)
    continue
```

Fig. 24: Menu Choice 1

If the user inputs 2, we'll display the data we have:

```
# Present the current data
elif menu_choice == "2":

    IO.output_student_courses(students)
    continue
```

Fig. 25: Menu Choice 2

If the user inputs 3, we'll save the data to a file:

```
# Save the data to a file
elif menu_choice == "3":

    FileProcessor.write_data_to_file(FILE_NAME, students)
    continue
```

Fig. 26: Menu Choice 3

If the user inputs 4, we'll stop the loop:

```
# Stop the loop
elif menu_choice == "4":
    break  # out of the loop
```

Fig. 27: Menu Choice 4

If anything else happens, we'll go back to the top–invalid menu choices are handled in the input_menu_choice function:

```
else:
    continue # on an error or invalid menu choice
```

Fig. 28: Else, Continue

And once we're out of the loop, we'll let the user know that the program is done running:

```
print("Program Ended")
```

Fig. 29: Program Ended

How satisfying! Every menu option is contained in a single line of code. This loop is now very straightforward and readable. Let's move on and test our code.

Testing

This week's code functions almost identically to last week's, so testing proceeds in a similar manner. We'll each menu option, check the saved data, and deliberately cause all expected errors. As with last week, the PyCharm and Command Prompt test sessions were sequential, and I've combined the testing screenshots from each. Any differences between the two will be noted when they occur. We can start by trying to run the code with no Enrollments.json file.

```
Text file must exist before running this script!

-- Technical Error Message --
[Errno 2] No such file or directory: 'Enrollments.json'
File not found.
<class 'FileNotFoundError'>
You can't close a file that doesn't exist!

-- Technical Error Message --
cannot access local variable 'file' where it is not associated with a value
Local name referenced but not bound to a value.
<class 'UnboundLocalError'>
```

```
Text file must exist before running this script!

-- Technical Error Message --
[Errno 2] No such file or directory: 'Enrollments.json'
File not found.
<class 'FileNotFoundError'>
You can't close a file that doesn't exist!

-- Technical Error Message --
cannot access local variable 'file' where it is not associated with a value
Local name referenced but not bound to a value.
<class 'UnboundLocalError'>
```

Fig. 30: No Such File

This causes two errors: one custom error for the file not existing, and an UnboundLocalError for trying to close a file that doesn't exist. But the program still runs! This tells us that our upgraded error handling works. Next, we can try and load a file that isn't in JSON format. Our dummy Enrollments.JSON file looks like this:

```
junk data
[{"FirstName": "Bob", "LastName": "Smith", "CourseName": "Python
100"}, {"FirstName": "Sue", "LastName": "Jones", "CourseName":
"Python 100"}]
```

Fig. 31: Junk Data

Running the program with this edit to the original file causes the JSON decoder to get stuck, and throws this error:

```
There was an unspecified error!

-- Technical Error Message --
Expecting value: line 1 column 1 (char 0)
Subclass of ValueError with the following additional properties:

msg: The unformatted error message
doc: The JSON document being parsed
pos: The start index of doc where parsing failed
lineno: The line corresponding to pos
colno: The column corresponding to pos


<class 'json.decoder.JSONDecodeError'>
```


```
There was an unspecified error!

-- Technical Error Message --
Expecting value: line 1 column 1 (char 0)
Subclass of ValueError with the following additional properties:

msg: The unformatted error message
doc: The JSON document being parsed
pos: The start index of doc where parsing failed
lineno: The line corresponding to pos
colno: The column corresponding to pos


<class 'json.decoder.JSONDecodeError'>
```


Fig. 32: Junk Data Error

The error even tells us where the decoder got stuck: at the very first character. Now, let's fix Enrollments.json and test the normal functions of the program. Initializing with a correctly named and formatted file brings us to the menu:

```
---- Course Registration Program ----
  Select from the following menu:
    1. Register a Student for a Course
    2. Show current data
    3. Save data to a file
    4. Exit the program
----------------------------------------

What would you like to do:
```

```
---- Course Registration Program ----
  Select from the following menu:
    1. Register a Student for a Course
    2. Show current data
    3. Save data to a file
    4. Exit the program
----------------------------------------

What would you like to do:
```

Fig. 33: Main Menu

First, we can try an invalid menu option, like 5.

```
What would you like to do: 5
Please choose a valid option!
```

```
What would you like to do: 5
Please choose a valid option!
```

Fig. 34: Invalid Menu Option

We're booted back to the main menu; great! Next, let's quickly test out what happens when we input an invalid first and last name.

```
What would you like to do: 1
Enter the student's first name: JOHN
The first name should contain only letters.

-- Technical Error Message --
The first name should contain only letters.
Inappropriate argument value (of correct type).
<class 'ValueError'>
```

```
What would you like to do: 1
Enter the student's first name: J0HN
The first name should contain only letters.

-- Technical Error Message --
The first name should contain only letters.
Inappropriate argument value (of correct type).
<class 'ValueError'>
```

Fig. 35: Invalid First Name

```
What would you like to do: 1
Enter the student's first name: JOHN
Enter the student's last name: SM1TH
The last name should contain only letters.

-- Technical Error Message --
The last name should contain only letters.
Inappropriate argument value (of correct type).
<class 'ValueError'>
```

```
What would you like to do: 1
Enter the student's first name: JOHN
Enter the student's last name: SM1TH
The last name should contain only letters.

-- Technical Error Message --
The last name should contain only letters.
Inappropriate argument value (of correct type).
<class 'ValueError'>
```

Fig. 36: Invalid Last Name

Looking good so far. Each of these inputs kicks us out to the main menu; we'll find out whether it saved anything when we check option 2. Now let's input a valid first and last name; note that like last week, I'm using different names for PyCharm and Command Prompt. This will let us test the save function in a convenient way.

```
What would you like to do: 1
Enter the student's first name: John
Enter the student's last name: Smith
Please enter the name of the course: Python 100
You have registered John Smith for Python 100.
```

```
What would you like to do: 1
Enter the student's first name: Alice
Enter the student's last name: Roberts
Please enter the name of the course: Python 100
You have registered Alice Roberts for Python 100.
```

Fig. 37: Valid First and Last Name

Looking great. Let's see what we have so far:

```
What would you like to do: 2
-------------------------------------------------
Bob, Smith, Python 100
Sue, Jones, Python 100
John, Smith, Python 100
-------------------------------------------------
```

```
What would you like to do: 2
-------------------------------------------------
Bob, Smith, Python 100
Sue, Jones, Python 100
John, Smith, Python 100
Alice, Roberts, Python 100
-------------------------------------------------
```

Fig. 38: Menu Option 2

Next, we can save our data.

```
What would you like to do: 3
The following data was saved to file!
Student Bob Smith is enrolled in Python 100
Student Sue Jones is enrolled in Python 100
Student John Smith is enrolled in Python 100
```

```
What would you like to do: 3
The following data was saved to file!
Student Bob Smith is enrolled in Python 100
Student Sue Jones is enrolled in Python 100
Student John Smith is enrolled in Python 100
Student Alice Roberts is enrolled in Python 100
```

Fig. 39: Menu Option 3

Now, we can exit the program and check Enrollments.json.

```
What would you like to do: 4
Program Ended
```

```
What would you like to do: 4
Program Ended
```

Fig. 40: Menu Option 4

```
[{"FirstName": "Bob", "LastName": "Smith", "CourseName": "Python 100"},
{"FirstName": "Sue", "LastName": "Jones", "CourseName": "Python 100"},
{"FirstName": "John", "LastName": "Smith", "CourseName": "Python 100"},
{"FirstName": "Alice", "LastName": "Roberts", "CourseName": "Python 100"}]
```

Fig. 41: Saved Data

The data in the text file matches what we saved during the Command Prompt test session, so we've confirmed that our code is fully functional.

GitHub

Because we're putting this version of the script in its own repo, the GitHub portion of this assignment was quick and simple. We created a new repository, uploaded all of the required files, committed the changes, and posted a link to GitHub on Canvas.

Summary

Following the trend of these lessons, classes and functions made my code longer but more usable. As the toolkit I'm using grows, it becomes more and more clear why these unified organizational methods are critical to collaboration. Even simply writing docstrings forced me to make absolutely sure that I understood what every piece of my code was doing, and why. My code didn't go through many functional revisions between the first and final drafts, but even the small tweaks I made felt like they had a huge impact on the consistency and readability of my code. And naturally, the prospect of reusability is exciting to me as a student–less time spent typing means more time spent digging into the interesting parts of the material!