Matthew Johnson

November 20, 2023

IT FDN 110 A

Assignment06

GitHub Link: https://github.com/jwehttam/IntroToProg-Python-Mod06

Python: Classes, Functions and Structured Error Handling

Introduction

In our latest assignment, we looked at three new ways to improve our code: Classes, Functions and Structured Error Handling. Classes offer a way of combining data and functionality, Functions make use of reusable code blocks, and Structured Error Handling provides predictable program behavior. This paper talks about these concepts, using examples from our Student Registration program.

Classes

I am thinking of Python classes to be like templates for making objects. They bundle together data and the functions that work with that data. They help promote code reuse and help make our code more modular.

Data Management

The FileProcessor class is used as a method for data interactions with files. It shows how classes can focus on certain tasks with data, helping to make code cleaner and simpler by keeping tasks separate from other parts of a program. This class defines how to read from, and write to our Enrollments.json file (Figure 1).

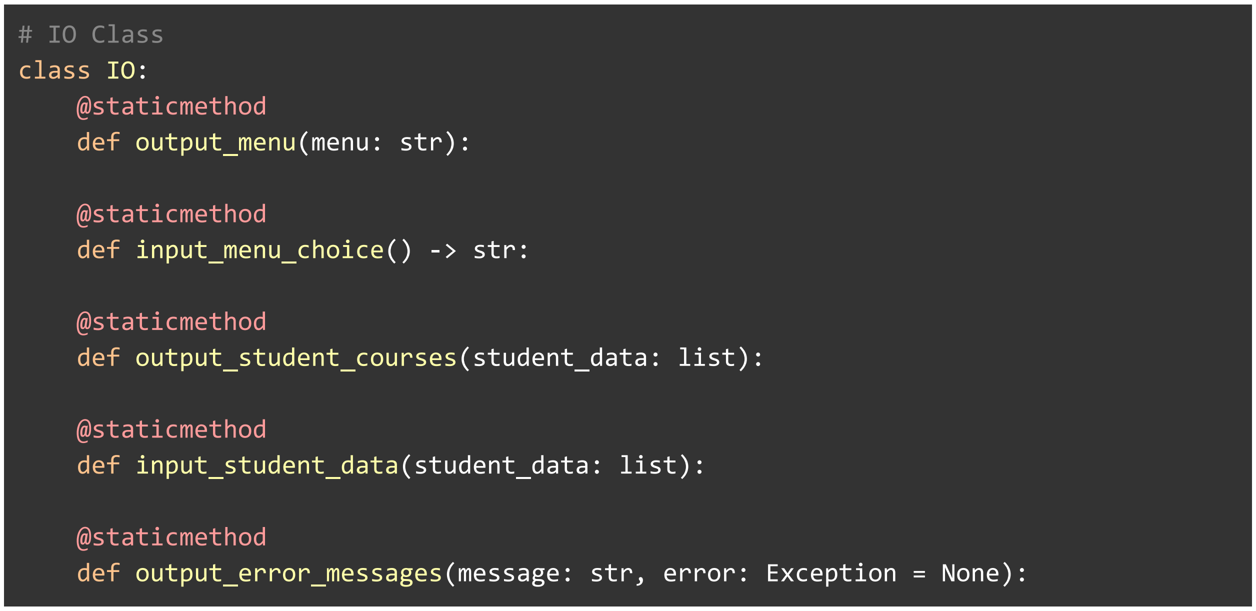
```
# FileProcessor Class
class FileProcessor:
    @staticmethod
    def read_data_from_file(file_name: str, student_data: list):

    @staticmethod
    def write_data_to_file(file_name: str, student_data: list):
```

Figure 1: FileProcessor Class

Modularity and Reusability

The I/O class contains static methods related to the user's interactions in the program. Again, this class groups related functions to incorporate modularity and code reuse. This includes displaying the menu, capturing the user's input, formatting the output for display, and exception handling (Figure 2).



```python
# IO Class
class IO:
    @staticmethod
    def output_menu(menu: str):

    @staticmethod
    def input_menu_choice() -> str:

    @staticmethod
    def output_student_courses(student_data: list):

    @staticmethod
    def input_student_data(student_data: list):

    @staticmethod
    def output_error_messages(message: str, error: Exception = None):
```

Figure 2: I/O Class

Functions

Functions in Python are self-contained modules of code used to accomplish certain tasks. Functions make code easier to read and understand, cut down on repeated code, and organize the program by grouping steps into reusable parts.

Static Methods

The FileProcessor class, (I think of it like a toolkit), has static methods ('read_data_from_file' and 'write_data_to_file') that are specialized tools within that toolkit. These static methods are functions that don't depend on any particular instance of the FileProcessor class. They are part of the toolkit, but can operate independently of any tool in that toolkit. All methods in Python are functions, but not all functions are methods. A function becomes a method when they are a part of a class.

Implementing User Interaction

The static methods ('input_menu_choice' and 'input_student_data') in the I/O class show how functions can handle the user's interactions in the program. It helps simplify the user input processes and helps to reduce the code's complexity (Figure 3).

This code:

```
# input_student_data function
@staticmethod
def input_student_data(student_data: list):
   """
   Prompts the user for student information and adds it to the provided data list.

   :param student_data: The list to which the new student data will be added.
   :raises ValueError: If the input for the first or last name contains
non-alphabetic characters.
   ChangeLog:
   - MatthewJohnson, 11.17.2023: Created Function.
   """
   try:
       student_first_name = input('Enter the student\'s first name: ')
       if not student_first_name.isalpha():
           raise ValueError('The first name should not contain numbers.')
       student_last_name = input('Enter the student\'s last name: ')
       if not student_last_name.isalpha():
           raise ValueError('The last name should not contain numbers.')
       course_name = input('Please enter the name of the course: ')
       student_data.append({'FirstName': student_first_name,
                            'LastName': student_last_name,
                            'CourseName': course_name})
       print(f'You have registered {student_first_name} {student_last_name} for
{course_name}.')
```

Can be called like this, for Menu Choice 1:

```
# menu_choice 1
elif menu_choice == '1':
   IO.input_student_data(students)
```

Figure 3: User Interaction and Menu_Choice 1

Structured Error Handling

Structured error handling in Python can help with building resilient programs. The use of try-except blocks helps make stronger programs that can handle unexpected errors smoothly, keeping the program running without issues.

Basic Exception Handling

With basic exception handling, we now have a function for outputting error messages (Figure 4).

```
# output_error_messages function
@staticmethod
def output_error_messages(message: str, error: Exception = None):
   """
   Outputs the provided message and optional exception information.

   :param message: The error message to be displayed.
   :param error: Optional. The exception object containing additional error
information.
   ChangeLog:
   - MatthewJohnson, 11.17.2023: Created Function.
   """
   print(message)
   if error:
       print('-- Technical Error Message --')
       print(error.__doc__)
       print(str(error))
```

Figure 4: Output_error_messages function

Keeping with the ('input_student_data') function example from (Figure 3), we have included this in our try-except block (Figure 5).

```
# input_student_data function
@staticmethod
def output_error_messages(message: str, error: Exception = None):
# Continued Code...
except ValueError as error:
   IO.output_error_messages('Invalid input', error)
except Exception as error:
   IO.output_error_messages('An unexpected error occurred', error)
```

Figure 5: Except block example

Customizing Error Messages

Customizing error messages is useful as a part of structured error handling. It means tailoring feedback to users when an error occurs by making it more specific and helpful. I'll use the 'inupt_student_data' static method as an example of this.

In this method, we are prompting the user to enter student information. We want to make sure that when registering, both the student's first and last names contain only alphabetic characters. If they enter a name that contains non-alphabetic characters, we raise a 'ValueError' and a customized message: 'The (first or last) name should not contain numbers.' Specific messages help inform the user precisely what went wrong.

This method includes a try-except block, so if a 'ValueError' is caught, the message is displayed using the 'IO.output_error_messages' function (Figure 6).

```
# Simplified Code Snippet
@staticmethod
def input_student_data(student_data: list):
    try:
        student_first_name = input('Enter the student\'s first name: ')
        if not student_first_name.isalpha():
            raise ValueError('The first name should not contain numbers.')
        student_last_name = input('Enter the student\'s last name: ')
        if not student_last_name.isalpha():
            raise ValueError('The last name should not contain numbers.')
    except ValueError as error:
        IO.output_error_messages('Invalid input', error)
    except Exception as error:
        IO.output_error_messages('An unexpected error occurred', error)
```
Figure 6: Customized ValueError Message

By customizing error messages, users are able to correct their inputs and move along with less confusion or frustration.

Summary

This paper explored the important concepts of Classes, Functions, and Structured Error Handling, using our Student Registration Program script as a reference. Understanding all of these elements is important, as they lay the foundation for writing code that is more modular and maintainable overall.