Name: Kelela Boreta

Date: 07 Aug 2024

Course: Foundations of Programming: Python

Assignment: Module 6 – Functions

https://github.com/kboreta/IntroToProg-Python-Mod06

Using Constants, Variables, and Print Statements

Introduction

This paper presents a comprehensive analysis of a Python script designed to handle student course registrations, focusing on the integration of classes, functions, and structured error handling. By breaking down the script, we aim to demonstrate how these programming constructs enhance the functionality, maintainability, and reliability of the software. This script serves as an educational tool, showcasing effective programming techniques in Python for managing user input, processing data, and interacting with files.

Script Breakdown

Script Header Description

The header of the Python script serves as a professional summary and documentation of the script's purpose and history. It is designed to provide clear and concise information about the script, including its title, description, and change log. This header follows a structured format that is commonly used in software development to ensure consistency and ease of understanding.

Title

The title of the script, "Assignment06," indicates that this program is part of a series of assignments or projects. It helps in quickly identifying the script in the context of other assignments, providing a clear reference point for instructors or peers reviewing the work.

Description

The description, "This assignment demonstrates using classes, functions, and structured error handling," outlines the primary focus and objectives of the script. It highlights the key programming concepts being applied, such as object-oriented programming through classes, the modular approach of using functions, and the implementation of structured error handling. This information is crucial for understanding the learning goals associated with the script and sets expectations for what the script aims to accomplish.

Change Log

The change log provides a chronological record of the modifications made to the script. It includes entries with three components:

- Who: Identifies the individual responsible for the changes. In this case, it begins with the initials "KBoreta" and later specifies "Kelela Boreta," ensuring that the contributions are traceable to a specific person.

- When: Records the date on which the changes were made. The entries “08/07/24” and “08/07/2024” document when significant modifications occurred, providing a timeline of development.
- What: Describes the nature of the changes. The initial entry notes the creation of the script, while the subsequent entry indicates modifications to include classes and functions, reflecting the script’s evolution.

Overall, the header serves as an essential tool for documentation, ensuring that the script’s purpose, history, and contributions are clearly communicated to anyone who interacts with the code.

```
#  ------------------------------------------------------------------------#
# Title: Assignment06
# Desc: This assignment demonstrates using classes, functions, and structured
error handling
# Change Log: (Who, When, What)
#   KBoreta,08/07/24,Created Script
# ------------------------------------------------------------------------#
```

Understanding "import json" in Python

The statement `import json` in Python is a fundamental part of handling JSON (JavaScript Object Notation) data within a Python program. JSON is a widely-used data interchange format that is easy for humans to read and write and easy for machines to parse and generate. It is a text format that is language-independent but uses conventions familiar to programmers of the C family of languages, including Python.

Purpose of the `json` Module

The `json` module in Python provides methods for parsing JSON data into Python objects and for converting Python objects into JSON format. This functionality is crucial for many applications, especially those involving web services and APIs, where JSON is commonly used as the format for data exchange.

Key Functions of the `json` Module

1. `json.load(file_object)`: This function parses a JSON file or file-like object into a Python dictionary. It reads the JSON content and converts it into corresponding Python data structures, such as lists and dictionaries. This is particularly useful when you need to load configuration files or data files in JSON format.

2. `json.loads(json_string)`: This function parses a JSON-formatted string and converts it into a Python dictionary or list. It's used when the JSON data is available as a string, such as data received from a web API.

3. `json.dump(data, file_object)`: This function serializes a Python object into a JSON-formatted string and writes it to a file or file-like object. It is used when you need to save data to a file in a structured and easily readable format.

4. `json.dumps(data)`: This function serializes a Python object into a JSON-formatted string. It's useful for converting data structures into a string format that can be transmitted over a network or stored in a database.

Importance in Python Programming

By importing the `json` module, Python programs gain the ability to work with JSON data seamlessly, allowing for easy data interchange between systems and platforms. This is particularly important in the context of web development, data analysis, and applications that require communication with external services

The use of `import json` in a Python script signifies that the program will interact with JSON data, either by reading from JSON files or strings, writing data in JSON format, or both. This capability enhances the program's interoperability and flexibility, making it an essential tool in modern software development.

```
import json
```

Constants and Variables

The script begins by defining two constants: `MENU` and `FILE_NAME`. The `MENU` constant is a formatted string that provides users with a list of options they can select from when interacting with the program. It acts as a user interface guide, helping users navigate through the program's features. The `FILE_NAME` constant is set to `"Enrollments.json"`, specifying the file used to store student enrollment data in JSON format.

Two primary variables are defined: `menu_choice`, which stores the user's menu selection, and `students`, an empty list that will hold student registration data as a collection of dictionaries.

```
# Define the Data 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"

# Define the Data Variables
menu_choice: str = ''
students: list = []
```

Class: FileProcessor

The `FileProcessor` class is responsible for managing file operations, specifically reading from and writing to the specified JSON file. It consists of two static methods:

- `read_data_from_file`: This method attempts to read student data from the file specified by `FILE_NAME`. It uses JSON parsing to load the data into the `students` list, handling any exceptions that might occur during the file read operation. The use of exception handling ensures that the program can provide informative feedback if an error arises, such as a missing file or invalid JSON format.

- `write_data_to_file`: This method writes the current state of the `students` list to the JSON file. It also includes error handling to manage potential issues during the file write operation. After successfully writing the data, it displays the current student registrations using the `IO.output_student_courses` method.

```
class FileProcessor:
    """Processes data to and from a file"""

    @staticmethod
    def read_data_from_file(file_name: str, student_data: list):
        """Reads data from a file into a list of dictionary rows"""
        try:
            with open(file_name, "r") as file:
                student_data.clear()
                student_data.extend(json.load(file))
        except Exception as e:
            IO.output_error_messages("Error: There was a problem with reading
the file.", e)

    @staticmethod
    def write_data_to_file(file_name: str, student_data: list):
        """Writes data from a list of dictionary rows to a file"""
        try:
            with open(file_name, "w") as file:
                json.dump(student_data, file)
            IO.output_student_courses(student_data)
        except Exception as e:
            IO.output_error_messages("Error: There was a problem with writing
to the file.", e)
```

Class: IO

The `IO` class manages interactions between the program and the user. It provides methods for input and output operations, ensuring that data is captured accurately and presented clearly:

- `output_error_messages`: This method displays error messages to the user, offering both a general error message and detailed technical information when available. This feature is essential for debugging and user support, allowing users to understand and resolve issues.

- `output_menu`: Displays the menu options to the user, guiding them through the available functionalities of the program.

- `input_menu_choice`: Captures and returns the user's menu choice as a string. It ensures that the program can respond to user selections.

- `output_student_courses`: Formats and displays the list of registered students and their courses. This method iterates over the `students` list, outputting each student's information in a readable format.

- `input_student_data`: Collects data for new student registrations, prompting the user for a first name, last name, and course name. The method includes validation checks to ensure that names are alphabetic

and course names are not empty. It handles exceptions related to input validation, providing user-friendly error messages if invalid data is entered.

```
class IO:
    """Performs Input and Output operations"""

    @staticmethod
    def output_error_messages(message: str, error: Exception = None):
        """Outputs error messages to the user"""
        print(message)
        if error:
            print("-- Technical Error Message -- ")
            print(error.__doc__)
            print(error.__str__())

    @staticmethod
    def output_menu(menu: str):
        """Displays the menu to the user"""
        print(menu)

    @staticmethod
    def input_menu_choice():
        """Gets the user's menu choice"""
        return input("What would you like to do: ").strip()

    @staticmethod
    def output_student_courses(student_data: list):
        """Displays the student data"""
        print("-" * 50)
        for student in student_data:
            print(f'Student {student["FirstName"]} {student["LastName"]} is
enrolled in {student["CourseName"]}')
        print("-" * 50)

    @staticmethod
    def input_student_data(student_data: list):
        """Gets student data from the user and adds it to the list"""
        try:
            first_name = input("Enter the student's first name: ").strip()
            if not first_name.isalpha():
                raise ValueError("The first name should not contain numbers
or be empty.")

            last_name = input("Enter the student's last name: ").strip()
            if not last_name.isalpha():
                raise ValueError("The last name should not contain numbers or
be empty.")

            course_name = input("Please enter the name of the course:
").strip()
            if not course_name:
                raise ValueError("The course name should not be empty.")

            student_data.append({
                "FirstName": first_name,
                "LastName": last_name,
```

```
                "CourseName": course_name
            })
            print(f"You have registered {first_name} {last_name} for
{course_name}.")
        except ValueError as e:
            IO.output_error_messages("Invalid input.", e)
        except Exception as e:
            IO.output_error_messages("Error: There was a problem with your
entered data.", e)
```

Main Program Loop

The script employs a `while` loop to continuously present the menu and process user inputs. This loop drives the core functionality of the program:

- Option 1: Register a Student for a Course: Invokes `IO.input_student_data` to add a new student entry to the `students` list.

- Option 2: Show Current Data: Calls `IO.output_student_courses` to display all current student registrations.

- Option 3: Save Data to a File: Executes `FileProcessor.write_data_to_file` to persist the student data to the JSON file.

- Option 4: Exit the Program: Breaks the loop and ends the program execution.

The program includes validation to ensure that only valid menu options (1-4) are accepted, prompting the user if an invalid choice is made.

Summary

This Python script exemplifies the use of classes, functions, and structured error handling to create a robust course registration program. By dividing responsibilities between the `FileProcessor` and `IO` classes, the script adheres to the separation of concerns principle, enhancing both readability and maintainability. The incorporation of error handling ensures that the program can gracefully manage unexpected situations, providing clear feedback to users. Overall, this script serves as a valuable educational example of effective programming practices in Python.