TASK

OOP – Modules

Visit our website

Introduction

WELCOME TO THE MODULES TASK

In this task, we are going to build on the design and testing principles we covered previously, and explore the complementary implementation techniques in Python.

Let’s get started!

SETTING UP PROFESSIONAL-GRADE PYTHON PROJECTS

Creating a professional-grade Python project involves establishing a robust foundation that ensures code quality, maintainability, and collaboration. This section covers best practices for setting up your Python projects, including the use of linters, requirements.txt files, virtual environments, and recommended file structures.

When beginning work on a Python project, it’s always a good idea to set up a virtual environment and a requirements.txt file (if one does not exist), as well as installing a linter. This allows you to create isolated environments for your projects, each with its dependencies, without affecting the system-wide Python installation, supporting your development workflow. Follow the guide entitled **Project Setup Guide**, included in the folder for this task, for detailed instructions on how to do so.

Remember to activate the virtual environment whenever you work on your project, to ensure that the dependencies you install are specific to your project and don't interfere with other projects or the system-wide Python installation.

Virtual Environments

Using virtual environments is crucial to isolate project dependencies and avoid conflicts with system-wide packages.

File Structure

Organise your project files in a clear and consistent structure. A common structure example follows.

```
project_name/
|-- src/
|   |-- module1/
|   |   |-- __init__.py
|   |   |-- module1.py
|   |-- module2/
|   |   |-- __init__.py
|   |   |-- module2.py
|-- tests/
|   |-- test_module1.py
|   |-- test_module2.py
|-- README.md
|-- requirements.txt
|-- .gitignore
```

This structure separates source code (`src/`) from tests (`tests/`) and includes a `README.md` file for documentation, `requirements.txt` for dependencies, and `.gitignore` to specify files and directories to be ignored by version control.

Dependency Management

Creating a `requirements.txt` file is a common practice in Python development to document and manage project dependencies. This file lists all the Python packages and their versions that your project depends on, making it easier for others to replicate your environment.

In Python development, managing project dependencies is crucial for ensuring consistent and reproducible environments across different systems. The `requirements.txt` file serves as a blueprint for these dependencies, making it easier to share and replicate your project.

To generate a `requirements.txt` file, we often use tools like `pip freeze`.

```
pip freeze > requirements.txt
```

This command displays the installed packages and their versions in the current environment. By redirecting this output to a file, you create a snapshot of your project's dependencies.

Remember, maintaining a `requirements.txt` file is a best practice in Python development. It not only helps collaborators and contributors understand the dependencies of your project, but also facilitates the process of setting up a consistent development environment. With tools like `pip freeze`, creating and updating this file becomes a straightforward task, contributing to the overall reliability and reproducibility of your Python projects.

Version Control

Utilise a version-control system, such as Git, to track changes, collaborate with others, and maintain a history of your project. Initialise a Git repository:

```
git init
```

Linting

Linters help enforce coding standards and identify potential issues early in development. Popular Python linters include Flake8 and pylint.

INTRODUCTION TO PEP 8 LINTING

Linting is a process of analysing code for potential errors, style violations, and other issues. It helps maintain a consistent and high-quality codebase. We use a Python linter called **Flake8**.

Let's assume you have a Python script named **example.py** with the following content:

```
# example.py

def add_numbers(a, b):
    result = a + b
    print(result)
```

Now, let's introduce an intentional error to demonstrate how linting can catch issues. Modify the script as follows:

```
# example.py

def add_numbers(a, b):
    result = a + b
    print(result)

# Intentional error: Using an undefined variable 'c'
print(c)
```

Now, if we run a linter like Flake8 on this code, we will see an error message indicating the issue. Flake8 might output something like:

```
example.py:9:6: F821 undefined name 'c'
```

Explanation

- `example.py:9:6` indicates the file (`example.py`), line (`9`), and column (`6`) where the issue is located.
- `F821` is a Flake8 error code that corresponds to the issue "`undefined name`".

In this case, the linter has detected that the variable `c` is used without being defined, which is a common type of error. Linters can catch various issues, including style violations, unused variables, and other potential bugs, helping developers identify and fix problems early in the development process.

ADDING A PEP 8 LINTER

A linter is a tool that analyses source code to flag programming errors, bugs, stylistic errors, and suspicious constructs. The main functions of a linter include:

- **Identifying syntax errors** – pointing out where code fails to conform to the programming language's structure and rules.
- **Detecting bugs** – flagging potential bugs such as infinite loops, unused variables, etc. before code is run.
- **Enforcing style rules** – checking adherence to specified style guide rules about spacing, variable naming, etc.
- **Improving readability** – Identifying difficult-to-read code structures that can be rewritten.
- **Detecting code smells** – flagging suspicious patterns that may indicate a deeper problem.
- **Security alerts** – raising issues that could pose security vulnerabilities if exploited.

To add PEP 8 linting to a Python project, you can use a tool like **Flake8**, a popular linting tool that checks your code against the style guide outlined in PEP 8. See the additional reading PDF entitled **Modular Testing and Implementation: Walkthrough Guide** (included in your folder for this task) for further instructions.

Here are some common PEP 8 violation types and their resolutions:

- **Indentation violation:** Use four spaces per indentation level.
- **Whitespace violation:** Avoid extraneous whitespace at the beginning or end of a line.
- **Line length violation:** Limit all lines to a maximum of 79 characters (72 for docstrings and comments).

- **Imports violation:** Imports should usually be on separate lines and should be grouped in the following order: standard library imports, related third-party imports, and local application-/library- specific imports.
- **Blank lines violation:** Use blank lines sparingly, especially within functions or methods.

DEFINING MODULES TO SEPARATE CONCERNS

Introduction to Python Modules

In Python, a module is a file containing Python definitions and statements. The file name is the module name with the suffix **.py** added. Modules allow you to organise your code into separate files, making it easier to manage and understand. They also enable code reuse and maintainability.

When structuring a Python project, defining modules (creating modules) for different concerns and responsibilities is a good practice. Modularising your code helps improve maintainability, readability, and reusability. Below is a general guide to how you might organise modules based on different concerns and responsibilities.

File Organisation in VS Code

In VS Code, a well-organised Python project could look like this:

```
my_project/
│
├── main.py
├── data_access.py
├── business_logic.py
├── user_interface.py
├── utilities.py
├── config.py
├── tests/
│   └── tests.py
└── constants.py
```

Main Module (main.py)

```
# main.py
from user_interface import start_application

if __name__ == "__main__":
 start_application()
```

Explanation

- This is the main entry point of the application.
- It imports the `start_application` function from the `user_interface` module.
- The `__name__ == "__main__"` condition ensures that the `start_application` function is called only when the script is executed directly, not when it is imported as a module.

Data Access Module (data_access.py)

```
# data_access.py
class TaskRepository:
 def get_tasks(self):
 """ Retrieve tasks from the data source."""
 # Placeholder implementation - replace with actual data retrieval
logic
 pass

 def save_task(self, task):
 """ Save a task to the data sink."""
 pass
```

Explanation

In this code, we implement a basic form of the repository pattern in Python.

- The repository pattern is a design pattern commonly used in software development to separate the logic that retrieves data from the underlying storage system (such as a database) from the rest of the application.
- This module defines a `TaskRepository` class responsible for handling data access operations.
- It includes methods `get_tasks` and `save_task` for retrieving tasks from, and saving tasks to, a data source, respectively.
- Docstrings provide a brief description of each method's purpose.

Business Logic Module (business_logic.py)

```
# business_logic.py
from data_access import TaskRepository

class TaskService:
 def __init__(self):
 """ Initialise the TaskService with a TaskRepository."""
 self.task_repository = TaskRepository()
```

```
def get_all_tasks(self):
""" Get all tasks from the data source."""
return self.task_repository.get_tasks()

def add_task(self, task):
""" Add a new task to the data source."""
self.task_repository.save_task(task)
```

Explanation

- This module defines a `TaskService` class that encapsulates the business logic of the application. (A service is called a "service" because it provides a service or a well-defined set of operations that can be utilised by other parts of the application.)
- It initialises a `TaskRepository` in the constructor to interact with the data source.
- The methods `get_all_tasks` and `add_task` use the `TaskRepository` to get all tasks, and add a new task, respectively.
- Docstrings provide explanations of the class and each method.

User Interface Module (user_interface.py)

```
# user_interface.py
from business_logic import TaskService

def start_application():
 """ Start the Task Management Application."""
 task_service = TaskService()

 while True:
  print("Task Management Application")
  print("1. View Tasks")
  print("2. Add Task")
  print("3. Quit")

  choice = input("Enter your choice: ")

  if choice == "1":
   tasks = task_service.get_all_tasks()
   print("Tasks:")
   for task in tasks:
    print(f"- {task}")
  elif choice == "2":
   new_task = input("Enter task description: ")
   task_service.add_task(new_task)
   print("Task added successfully!")
  elif choice == "3":
   print("Exiting the application. Goodbye!")
   break
  else:
   print("Invalid choice. Please try ag
```

Explanation

- This module contains the user interface logic for the application.
- The `start_application` function initialises a `TaskService`.
- It provides a simple command-line interface for users to view tasks, add tasks, or quit the application.
- The user's input determines the action to be taken, and the corresponding methods of `TaskService` are called.
- Docstring explains the purpose of the `start_application` function.

Utilities Module (utilities.py)

```
# utilities.py
def format_date(date):
 """ Format a date string."""
 pass
```

Explanation

- This module defines a utility function, `format_date`, for formatting date strings.
- The function can be used across the application for consistent date formatting.
- The docstring provides a brief description of the utility function's purpose.

Configuration Module (config.py)

```
# config.py
```

```
FILENAME = "data.txt"
```

Explanation

- This module holds configuration settings for the application.
- `FILENAME` is an example configuration parameter representing the filename for the file we will back our data to.
- In a real-world scenario, this module could contain a variety of configuration settings for a database connection. For the scope of this task, we will focus on the filename.

Tests Module (tests.py)

```
# tests.py
import unittest
from business_logic import TaskService, Task

class TestTaskService(unittest.TestCase):
    def test_add_task(self):
        """Test the add_task method of TaskService."""
        # Arrange
        task_service = TaskService()
        initial_task_count = len(task_service.get_all_tasks())  # Get
initial task count
        new_task = Task(title="New Task", description="Description of
the new task")

        # Act
        task_service.add_task(new_task)

        # Assert
        updated_task_count = len(task_service.get_all_tasks())  # Get
updated task count
        self.assertEqual(updated_task_count, initial_task_count + 1)  #
Check if the task count increased by 1
        self.assertIn(new_task, task_service.get_all_tasks())  # Check
if the new task is in the list of tasks

if __name__ == "__main__":
    unittest.main()
```

Explanation

- **Arrange:** Set up the necessary objects and conditions for the test. In this case, create an instance of `TaskService`, get the initial task count, and create a new task.
- **Act:** Perform the action you are testing. In this case, call the `add_task` method with the new task.
- **Assert:** Check whether the actual result matches the expected result. Here, we check whether the task count has increased by one and whether the new task is in the list of tasks.
- The docstrings explain the purpose of each test case. The Arrange-Act-Assert pattern, also called Given-When-Then, is a commonly used structure for tests. Tests make it easier to detect bugs early. Writing good tests comes with practice, so it is a good idea to write them in all your future tasks, from this point forward, to build proficiency.

Constants Module (constants.py)

```
# constants.py
TASK_MAX_LENGTH = 100
```

Explanation

- This module defines constant values that can be used across the application.
- `TASK_MAX_LENGTH` is an example constant representing the maximum length allowed for a task description.

Now let's put some of this new knowledge into practice!

Instructions

Read and run the code in the additional reading entitled **Project Setup Guide** to become more comfortable with the concepts covered in this task.

Practical Task

In this practical task, you are going to implement and test the task manager application you designed in the Software Design Task.

Follow the steps below to complete the task. You do not have to provide written answers to the questions, as they are meant to guide your thinking.

1) Set up a virtual environment for your project while considering the following points:
   - What steps will you take to create a virtual environment for your project?
   - Which tool or package will you use to manage the virtual environment?
2) PEP 8 Linting: Set up a PEP 8 linter for your project, considering the below points:
   - How will you integrate PEP 8 linting into your project?
   - Are there specific PEP 8 rules or configurations you want to enforce or ignore?
3) PEP 8 Compliance:
   - Ensure that your code adheres to PEP 8 standards.
   - Make any necessary adjustments to achieve compliance.
4) Implement your task manager application following your design:
   - Develop the necessary classes based on your design. Feel free to redesign the application to suit your implementation needs if you have discovered design improvements during your implementation phase.
   - Implement a file-based data access layer to support your application.
   - Remember to split your code into multiple modules. A common practice is to have one class or set of related functions or constants per Python module.

5) Write unit tests:
   - Identify at least five use cases based on your design task.
   - Write unit tests that test your implementation against these use cases. We suggest simplifying the tests you choose by testing the model, because testing the code with dependencies on a file and terminal will require that the file or terminal be available at the time that you run your tests. You cannot guarantee their availability, and this would complicate your workflow. If you are interested in how such tests are performed, you can read about mock testing and integration testing in your spare time.
6) Use a tool like `pip freeze` to generate a requirements.txt file.

Rate us

Share your thoughts

HyperionDev strives to provide internationally excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved or think we've done a good job?

Click here to share your thoughts anonymously.