Hyperiondev

Task

OOP – Modules

Visit our website

Introduction

Welcome to the OOP – Modules task!

In this task, we will explore implementation techniques in Python, focusing on setting up professional-grade projects, managing dependencies, and adhering to best practices like PEP 8 linting. You'll learn how to organise your code into modules, create virtual environments, and ensure your project is maintainable and collaborative. 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. The **requirements.txt** file specifies project dependencies and their versions, ensuring consistent installation across environments. By creating isolated environments for your projects, you don't need to rely on the system-wide Python installation. Each project can have its own set of dependencies, independent of the system's Python setup. For detailed instructions, please refer to the **Project Setup Guide** included in the task folder.

Virtual environments

Using virtual environments is crucial for isolating project dependencies and avoiding conflicts with system-wide packages. 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 that they don't interfere with other projects or the system-wide Python installation.

File structure

Organise your project files into a clear and consistent structure. A common structure example is the following:

```
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 `README.md` for documentation, `requirements.txt` for dependencies, and `.gitignore` that specifies 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.

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

You should use a version-control system, such as Git, to track changes, collaborate with others, and maintain a history of your project.

Reminder: A Git repository can be initialised using the following command:

```
git init
```

Linting

Linting is the process of analysing code to flag programming errors, bugs, stylistic errors, and other issues. Linters enforce coding standards and help identify potential problems early in development, ensuring code quality and consistency. By using linters, developers can maintain clean, readable, and maintainable code.

Pep 8 linting

Introduction to PEP 8 linting

As mentioned, linting is a process of analysing code for potential errors, style violations, and other issues, helping maintain a consistent and high-quality codebase.

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

```
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:

```
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:7:7: F821 undefined name 'c'
```

Explanation:

- `example.py:7:7` indicates the file (**example.py**), line (7), and column (7) 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 like this, including style violations, unused variables, and other potential bugs.

Functionality of a PEP 8 linter

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** that we introduced above, which is a popular linting tool that checks your code against the style guide outlined in PEP 8. See the additional **Project Setup 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, define modules for different concerns and responsibilities. This modularisation enhances maintainability, improves readability, and promotes reusability by logically organising and isolating code. 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)

```python
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)

```
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)

```
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)


class Task:
    def __init__(self, title, description):
        """Initialises a Task object with a title and description"""
        self.title = title
        self.description = description
```

Explanation:

- This module defines a `TaskService` class that encapsulates the business logic of the application. It's 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.
- Additionally, this module defines a `Task` class, which represents a task with a title and description. The `Task` class is used to create task objects that can be managed by the `TaskService`.
- Docstrings provide explanations of each class and each method.

User interface module (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 again.")
```

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.
- The docstring explains the purpose of the `start_application` function.

Utilities module (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)

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

Explanation:

- This module holds configuration settings for the application.
- `FILENAME` represents a configuration variable that specifies the filename where the application stores its data.
- 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)

```
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's a good idea to write them in all your future tasks from this point forward to build proficiency.

Take note:

The test will fail until you complete the logic for the `TaskRepository` class in the **data_access.py** module. The `get_tasks` method currently does not retrieve any data, and the `save_task` method also needs logic added to store tasks. You will need to update these methods to successfully retrieve and store tasks based on your data storage approach.

Constants module (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 **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 by following these steps:
    - Determine the steps that are required to create and activate a virtual environment for your project.
    - Select a tool for managing the virtual environment (such as `venv`).

2. Integrate PEP 8 linting into your project:
    - Decide how you will integrate PEP 8 linting.
    - Specify any particular PEP 8 rules or configurations you want to enforce or ignore through the use of configuration files such as **.flake8**.

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 that follows your design:
    - Develop the necessary classes based on your design. Feel free to update the designs for 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 read from and write to files for storing data. This ensures that the application's data is safely stored and easily accessible, even after the application is closed.

- 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 four different use cases** based on your designs for the task management application. You may need to redesign the application to include at least four use cases.
    - Create unit tests for each identified use case. These tests should verify that your code functions as expected under different scenarios:
        - Write unit tests that focus on the core logic and data structures of your application.
        - Avoid testing parts of the application that depend on external systems, such as text files. This can simplify your tests while making them more reliable.

6. Use a tool like `pip freeze` to generate a **requirements.txt** file.
    - Depending on your project's dependencies and whether you've installed any additional packages via `pip`, the **requirements.txt** file may be empty if no external packages are required for your project.

Important: Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.

Share your thoughts

Please take some time to complete this short feedback **form** to help us ensure we provide you with the best possible learning experience.