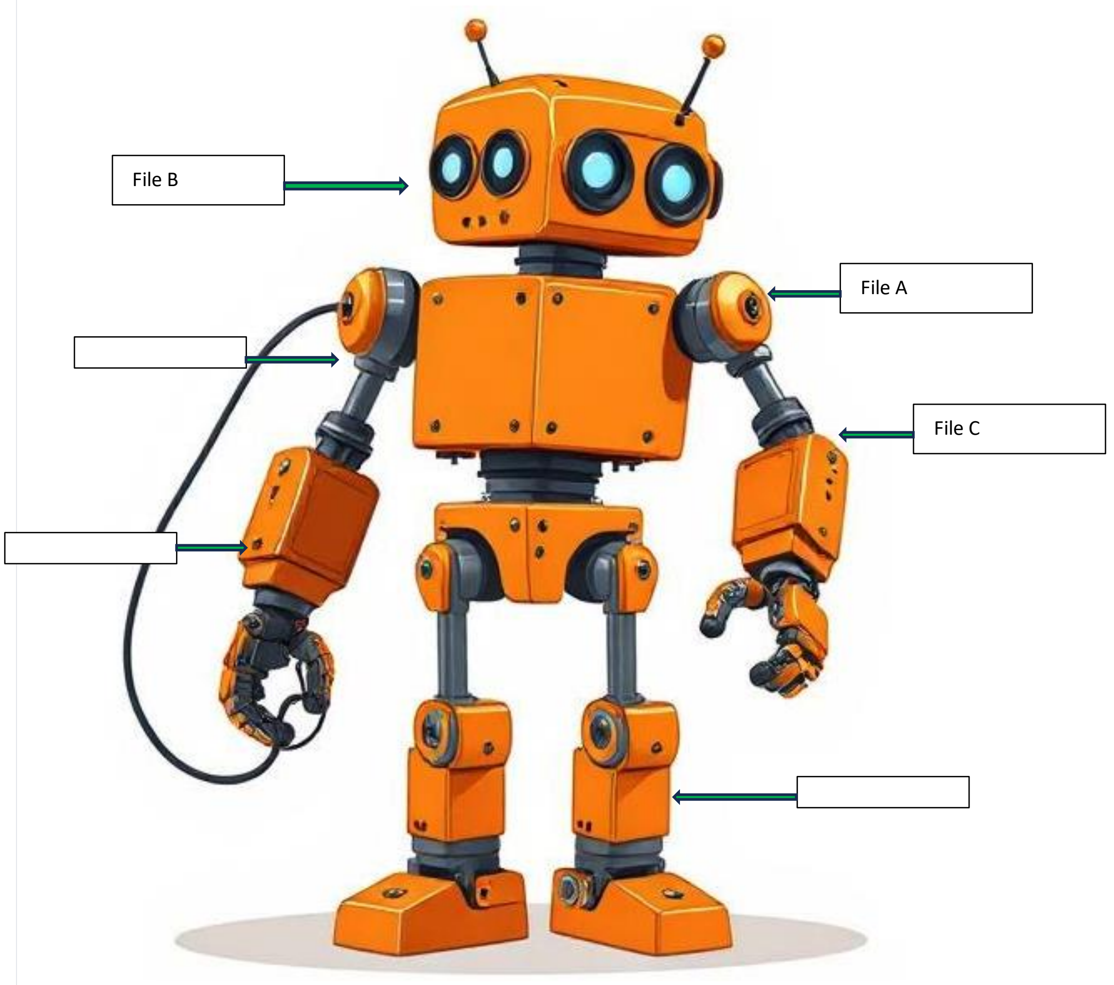
Benefits of Modular architecture:

1- **Maintainable**
2- **Scalable**
3- **Flexible**
4- **Designed for future growth**

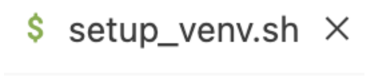


1. Create a Project Structure

First, let's establish a standard project structure:

```
my_project/
├── my_package/
|   ├── __init__.py
|   ├── module1.py
|   ├── module2.py
|   └── subpackage/
|       ├── __init__.py
|       └── submodule.py
├── tests/
|   ├── __init__.py
|   ├── test_module1.py
|   └── test_module2.py
├── docs/
|   └── documentation.md
├── .gitignore
├── README.md
├── requirements.txt
└── setup.py
```

2. Set Up a Virtual Environment

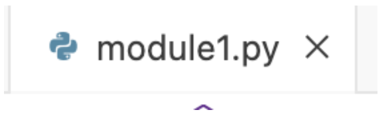


```
# Create a virtual environment
python -m venv venv

# Activate the virtual environment
# On Windows
# venv\Scripts\activate
# On macOS/Linux
source venv/bin/activate

# Install dependencies
pip install -r requirements.txt

# Update pip
pip install --upgrade pip
```

3. Initialize Your Package

```
"""
My Package - A brief description of what your package does.

This package provides functionality for...
"""

__version__ = '0.1.0'
```

4. Create Module Files


module1.py

```python
"""
Module for handling data processing operations.

This module contains functions and classes for processing various types of data.
"""

class DataProcessor:
    """A class for processing data."""

    def __init__(self, data=None):
        """Initialize the DataProcessor with optional data."""
        self.data = data or []

    def process(self):
        """Process the data and return results."""
        if not self.data:
            return None

        # Example processing
        return [item * 2 for item in self.data]


def helper_function(input_data):
    """A helper function that can be imported separately."""
    return f"Processed: {input_data}"


if __name__ == "__main__":
    # Code that runs when this module is executed directly
    processor = DataProcessor([1, 2, 3])
    print(processor.process())
```

5. Create a Main Application File

main.py 1

```python
"""
Main application entry point.

This module ties together functionality from other modules.
"""

from my_package.module1 import DataProcessor, helper_function
from my_package.module2 import format_output


def main():
    """Main function that runs the application."""
    # Get some data (in a real app, this might come from user input or a file)
    data = [1, 2, 3, 4, 5]

    # Process the data
    processor = DataProcessor(data)
    result = processor.process()

    # Use a helper function
    processed_text = helper_function("sample text")

    # Format and display results
    formatted_result = format_output(result)

    print(f"Processed data: {formatted_result}")
    print(processed_text)

    return result


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

6. Create a Second Module

module2.py

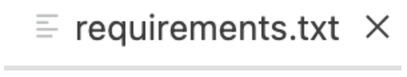
```python
"""
Module for output formatting and display.

This module contains functions for formatting and displaying output.
"""

def format_output(data):
    """Format data for display."""
    if not data:
        return "No data to display"

    if isinstance(data, list):
        return ", ".join(str(item) for item in data)

    return str(data)


def display_table(data, headers=None):
    """Display data in a tabular format."""
    if not data:
        return "No data to display"

    if not headers and isinstance(data[0], dict):
        headers = list(data[0].keys())

    # Simple table formatting logic
    result = ""
    if headers:
        result += " | ".join(headers) + "\n"
        result += "-" * (len(" | ".join(headers))) + "\n"

    for row in data:
        if isinstance(row, dict):
            result += " | ".join(str(row.get(h, "")) for h in headers) + "\n"
        else:
            result += str(row) + "\n"

    return result
```

7. Create a Test File

test_module1.py 1 ×

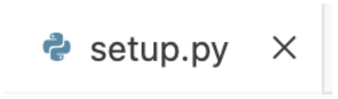
```python
"""
Tests for module1.py
"""

import pytest
from my_package.module1 import DataProcessor, helper_function


def test_data_processor_empty():
    """Test DataProcessor with empty data."""
    processor = DataProcessor()
    assert processor.process() is None


def test_data_processor_with_data():
    """Test DataProcessor with sample data."""
    processor = DataProcessor([1, 2, 3])
    result = processor.process()
    assert result == [2, 4, 6]


def test_helper_function():
    """Test the helper function."""
    result = helper_function("test")
    assert result == "Processed: test"
    assert isinstance(result, str)


if __name__ == "__main__":
    # Run tests manually if needed
    pytest.main(["-v", __file__])
```

8. Create a Requirements File


requirements.txt

```
# Core dependencies
pytest==7.3.1
black==23.3.0
flake8==6.0.0

# Optional dependencies depending on your project
# requests==2.30.0
# pandas==2.0.1
# matplotlib==3.7.1
```

9. Create a Setup File



```
from setuptools import setup, find_packages

setup(
    name="my_package",
    version="0.1.0",
    packages=find_packages(),
    install_requires=[
        # List your dependencies here
        "pytest",
    ],
    author="Your Name",
    author_email="your.email@example.com",
    description="A short description of your package",
    keywords="sample, package, python",
    url="https://github.com/yourusername/my_package",
    classifiers=[
        "Development Status :: 3 - Alpha",
        "Intended Audience :: Developers",
        "Programming Language :: Python :: 3",
        "Programming Language :: Python :: 3.8",
        "Programming Language :: Python :: 3.9",
    ],
    python_requires=">=3.8",
)
```

VS Code Configuration

1. Workspace Settings

VS Code workspace settings

```json
{
  "python.linting.enabled": true,
  "python.linting.flake8Enabled": true,
  "python.linting.pylintEnabled": false,
  "python.formatting.provider": "black",
  "python.formatting.blackArgs": [
    "--line-length",
    "88"
  ],
  "editor.formatOnSave": true,
  "python.testing.pytestEnabled": true,
  "python.testing.unittestEnabled": false,
  "python.testing.nosetestsEnabled": false,
  "python.testing.pytestArgs": [
    "tests"
  ],
  "[python]": {
    "editor.codeActionsOnSave": {
      "source.organizeImports": true
    }
  },
  "python.analysis.extraPaths": [
    "${workspaceFolder}"
  ]
}
```

2. Launch Configuration

VS Code launch configuration

launch.json 2

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Python: Current File",
      "type": "python",
      "request": "launch",
      "program": "${file}",
      "console": "integratedTerminal",
      "justMyCode": true
    },
    {
      "name": "Python: Main Module",
      "type": "python",
      "request": "launch",
      "module": "my_package.main",
      "console": "integratedTerminal",
      "justMyCode": true
    }
  ]
}
```

Working with Multiple Files in VS Code

1. Importing Between Modules

When working with multiple files, you'll import functionality from one module to another:

```
# In main.py
from my_package.module1 import DataProcessor
from my_package.module2 import format_output

# Use the imported components
processor = DataProcessor([1, 2, 3])
result = processor.process()
formatted = format_output(result)
```

2. Using VS Code's Navigation Features

- **Go to Definition**: Right-click on a class or function name and select "Go to Definition" or press F12
- **Find All References**: Right-click and select "Find All References" or press Shift+F12
- **Peek Definition**: Right-click and select "Peek Definition" or press Alt+F12

3. Using VS Code's Explorer View

- Group related files in folders
- Use the Explorer view to navigate between files
- Use "Split Editor" to view multiple files side by side

Best Practices for Multi-File Projects

1. **Follow the Single Responsibility Principle**: Each module should have a single responsibility
2. **Use meaningful names** for modules, classes, and functions
3. **Keep modules small** and focused on specific functionality
4. **Document your code** with docstrings and comments
5. **Use relative imports** within your package
6. **Create init.py files** in each directory to make them packages
7. **Use a consistent coding style** (configure Black and Flake8)
8. **Write tests** for each module

Example Workflow

1. Start a new project by creating the directory structure
2. Set up a virtual environment
3. Initialize Git repository (if using version control)
4. Create your package's `__init__.py` file

5. Create module files with specific functionality
6. Create a main application file that imports from modules
7. Write tests for your modules
8. Configure VS Code settings for the project

This organization approach will help you maintain clean, modular code that's easy to understand, test, and extend.