1. Appropriately package a Python code development.

Python is a versatile programming language used for various purposes, such as creating websites, industrial robots, games, etc. Its flexibility makes it essential to consider the project's target audience and environment before starting any Python project. Although it may seem unusual to consider packaging before coding, it helps prevent future issues (packaging.python.org, N.D.).

Programmers use packages to organise several packages and modules into an organised hierarchy. Similar to how drives and folders in an operating system assist us in storing files, this enables quick and simple access to directories and modules. Users can easily access sub-packages and modules by placing them within packages (GeeksforGeeks, 2017).

According to Gunn (2023), modular programming is splitting up big Python programmes into smaller modules utilising functions, modules, and packages. This method makes the project easier to manage and maintain when working in teams.

The Python community is its greatest asset. A package is already created and deployed by committed developers for every use case. Thanks to this level of dedication, Python is incredibly adaptable and perfectly suited to address unique demands (Briggs, 2021).

The following procedures should be followed to package a Python code development correctly, according to Gunn (2023):

1.1. Organising the code properly when working on a project is important. To do this, create separate folders for different project parts, such as code, data, testing, and documentation. Organising the code will help find what is needed quickly and make it easier for others to navigate it. Also, you can use consistent naming for files and folders throughout the project, following conventions like underscores for variables and functions and capital letters for classes.

1.2. Using version control like Git is also essential, even when working alone. It enables recording changes and backing up the work to a cloud-based repository. Many cloud-based Git solutions offer a free tier for solo practitioners. To manage dependencies, use a package manager like Pip. This is crucial when working with large projects with many dependencies.

1.3. Creating a virtual environment will isolate the project from others on the computer, preventing conflicts between packages used in different projects. Adding comments explaining what the code is doing and how to use it is important for making it more accessible to others. It also helps to remember your thoughts when writing the code.

1.4. Automated tests are essential for catching bugs early on. It can be used to check that the code works as expected. Using tools like Ruff or Flake8 to ensure the code looks consistent and catches common mistakes will also help to write better code. Finally, using a tool like setup tools to package and distribute the Python code will make it easier to share the work with others and ensure they can use the code without encountering any issues.

However, here is an example of how to create a package structure according to packaging.python.org. (N.D.):

- Creating the package files
- Creating a test directory
- Creating a test directory
- Creating pyproject.toml
- Configuring metadata
- Creating README.md
- Creating a LICENSE
- Generating distribution archives
- Uploading the distribution archive

Once the project is completed, the structure will appear like this:

```
packaging_tutorial/
├── LICENSE
├── pyproject.toml
├── README.md
├── src/
│   └── example_package_YOUR_USERNAME_HERE/
│       ├── __init__.py
│       └── example.py
└── tests/
```

(packaging.python.org.,N.D.)

2. Adequately and effectively document your code.

Proper documentation is essential in software creation. It aids stakeholders in using and maintaining code, as well as in the onboarding of new developers. Neglecting documentation may lead to repeated errors, yet many developers must pay more attention to its importance (Giacomelli, 2023).

Zadka (2019) states that Python code can contain embedded documentation by enclosing docstrings in triple quotes. More timely documented code can be avoided with proper documentation.

According to Mertz (2018), there are different ways to document Python code, but some of the common ones are:

Commenting the code: To ensure a clear understanding and efficient operation of the code, it is imperative to incorporate concise explanatory notes using the # symbol for specific lines or blocks of code.

Writing docstrings: Docstrings summarise an object's purpose for new users. They should be brief and clear for easy maintenance. PEP 257 explains the conventions. When defining them, it is imperative to include succinct descriptions of a module, function, class, or method. These descriptions, referred to as "docstrings," encapsulated in triple quotes ("""), should cover the object's purpose, parameters, return values, and potential exceptions.

Using documentation tools: Numerous external tools can generate HTML pages or other documentation formats from the code and docstrings. Sphinx, Epydoc, Read The Docs, Doxygen, MkDocs, pycco, and doctest are some of the most notable. Using these tools is crucial for producing well-structured, all-encompassing, and user-friendly documentation for the project.

3. Use the tools available in Python to test code quality.

Maintaining consistent code quality is crucial for the health and progress of large projects. However, it can be challenging to achieve this manually. Fortunately, several Python code tools simplify the task (Eriksen, 2019).

Furthermore, VanTol (2018) also mentioned that writing high-quality code is crucial for functionality, readability, and maintainability. Tools like PEP8 and linters improve consistency and identify issues early. Linters can also save time during code reviews and offer customisable options. Improving code quality is a gradual process that starts with awareness of its importance.

Python is well renowned for its simplicity of use in web development and test automation, and the Python testing framework is a dynamic framework built on Python. Excellent testing tools are therefore required due to these changes throughout time. Automated testing is supported by some Python frameworks and packages (Bose & Makwana, 2023). Here is a sample list of the best Python testing frameworks:

1. Behave, one of the most popular Python test frameworks, is renowned for helping behaviour-driven development (BDD). This framework and Cucumber are fairly similar. All test scripts are created in a straightforward language and are then added to the running code. Relevant requirements determine code behaviour. Behave enables the reuse of previously defined steps in other use-case scenarios.

2. Lettuce, a framework for automating BDD test structures based on Python and Cucumber, is another efficient option. It streamlines typical processes and makes BDD simpler to complete.

3. Robot Framework – This framework is mostly appropriate for acceptance testing. Although it was created in Python, it can also function on IronPython (.net-based) and

Jython (Java-based). Linux, macOS, and Windows are all compatible with the Robot Framework.

4. One of the most well-known open-source testing frameworks, Pytest offers API tests, unit tests, and functional tests.

5. TestProject is another open-source automation framework that supports the Pytest and Unittest frameworks and offers cloud-based and local HTML reporting and simple test automation programming.

6. The basic Python testing framework, PyUnit (Unittest), is default included with the Python package. Most programmers begin their testing with this.

7. The unit testing framework Testify is used for system and integration testing. It aims to replace and improve upon the well-known Unittest and Nose frameworks.

8. The Python programming language's standard library contains a module called Doctest that makes it simple to create tests based on the output of the default Python interpreter shell. It looks for interactive Python sessions to determine whether they are functioning properly.

References:

packaging.python.org. (N.D.). *An Overview of Packaging for Python — Python Packaging User Guide*. [online] Available at: https://packaging.python.org/en/latest/overview/.

GeeksforGeeks. (2017). *Create and Access a Python Package*. [online] Available at: https://www.geeksforgeeks.org/create-access-python-package/.

Gunn, E. (2023). *Best Practices in Structuring Python Projects | Dagster Blog*. [online] Available at: https://dagster.io/blog/python-project-best-practices [Accessed 4 Jul. 2023].

Briggs, J. (2021). *How to Create Python Packages | Towards Data Science*. [online] Medium. Available at: https://towardsdatascience.com/how-to-package-your-python-code-df5a7739ab2e [Accessed 4 Jul. 2023].

Giacomelli, J. (2023). *Documenting Python Code and Projects*. [online] Available at: https://testdriven.io/blog/documenting-python/.

Zadka, M. (2019). *How to document Python code with Sphinx | Opensource.com*. [online] opensource.com. Available at: https://opensource.com/article/19/11/document-python-sphinx [Accessed 6 Jul. 2023].

Mertz, J. (2018). *Documenting Python Code: A Complete Guide – Real Python*. [online] realpython.com. Available at: https://realpython.com/documenting-python-code/#documentation-tools-and-resources [Accessed 6 Jul. 2023].

Eriksen, M. (2019). 5 Awesome Tools for Python Code Quality. [online] www.madelyneriksen.com. Available at:

https://www.madelyneriksen.com/blog/awesome-python-code-quality-tools/#:~:text=5%20Awesome%20Tools%20for%20Python%20Code%20Quality%201 [Accessed 6 Jul. 2023].

VanTol, A. (2018). *Python Code Quality: Tools & Best Practices*. [online] Realpython.com. Available at: https://realpython.com/python-code-quality/.

Bose, S. & Makwan, A. (2023). *Top 8 Python Frameworks for Test Automation*. [online] Available at: https://www.browserstack.com/guide/top-python-testing-frameworks [Accessed 6 Jul. 2023].