ESA ESTEC
Keplerlaan 1
2201 AZ Noordwijk
The Netherlands

GUIDELINES FOR PYTHON DEVELOPMENT

Prepared by	EOP-PEP team
	ESA
Document Type	TN - Technical Note
Reference	PE-TN-ESA-GS-0872
Issue/Revision	0 . 2
Date of Issue	18/09/2024
Status	Draft

APPROVAL

Title	Guidelines for Python development		
Issue Number	0	Revision Number	2
Author	EOP-PEP team	Date	18/09/2024
Approved By		Date of Approval	

CHANGE LOG

Reason for change	Issue Nr	Revision Number	Date
Initial version	0	1	25/03/2024
Second revision	0	2	18/09/2024

CHANGE RECORD

Issue Number 0	Revision Number	1	
Reason for change	Date	Pages	Paragraph(s)
Initial version	25/03/2024		

Issue Number 0	Revision Number	2	
Reason for change	Date	Pages	Paragraph(s)
Second revision following contributions and comments by EOP-PEP team	18/09/2024		

DISTRIBUTION

Name/Organisational Unit

Table of Contents

1. INTRODUCTION

Python is a widely used and versatile programming language that offers many benefits such as readability, simplicity, and a rich set of libraries. However, to ensure the quality, consistency, and maintainability of Python code, it is important to follow some best practices and standards. This document provides a set of guidelines for Python development, covering the following topics:

- Code writing and development workflow (tools and environment)
- Performance optimization
- Delivery (packaging, versioning, artifact management)
- Recommended python packages for most common development areas

This document, designed to be applied to projects of medium to big size, aims to assist Python developers in creating code that is easier to maintain and improve performance. It also seeks to unify Python development and software products, as much as possible, for the ESA Earth Observation Directorate for activities related to Payload Data Ground Segments (especially Data Processors) and for End-to-end Mission Performance Simulator Chains while ensuring the best use of common features.

The document is based on the official Python documentation, the PEP (Python Enhancement Proposal) standards, and the best practices and conventions followed by the Python community. The document assumes that the reader has a basic knowledge of Python and its syntax and is familiar with the common tools and packages mentioned. The document also provides links and references to more detailed and specific resources for further learning and exploration.

1.1. Reference Documents

The following table specifies the reference documents that, while not binding, provide additional information.

Table 1: Reference documents

Reference	Code	Title	Issue
[E2E-ICD]	PE-ID-ESA-GS-464	ESA Generic E2E Simulator ICD	1.4.2
[PEP8]	https://peps.python.org/pep-0008/	PEP 8 – Style Guide for Python Code	
[PEP257]	https://peps.python.org/pep-0257/	PEP 257 – Docstring Conventions	
[PEP427]	https://peps.python.org/pep-0427/	PEP 427 – The Wheel Binary Package Format 1.0	

2. DEVELOPMENT WORKFLOW

A good development workflow is essential for writing, testing, debugging, and deploying Python code efficiently and effectively. A development workflow typically involves the following steps:

1. Organising the source code and packaging the application.
2. Setting up a virtual environment to isolate the project dependencies from the system-wide packages and installing and managing the required packages and libraries for the project.
3. Writing, formatting, and documenting the code following the Python style guide [PEP8] and docstring conventions compatible with an automatic documentation generator unique for all the Python deliverables (e.g. compatible with the Sphinx documentation generator such as NumPy style docstring format, Google style or reStructuredText with Sphinx extensions).
4. Running and debugging the code using an interactive interpreter, a code editor, or an integrated development environment (IDE).
5. Testing the code within the whole project using a single testing framework and tools, such as `unittest`, `pytest`, or `nose2`.

6. Versioning and tracking the changes in the code using Git as a version control system.
7. Distributing and deploying the code as a package, a module, a script, or an executable file, depending on the intended use and audience.

There are different tools and methods that can be used to implement each step of the development workflow. In this document, we will give an overview of the most common tools and provide recommendations on which ones to use in different contexts.

2.1. Organising the source code and packaging the application

One of the first steps in developing a Python application/package is to organise the source code in a way that is easy to maintain, test, and distribute. There are two common layouts for organising the source code: flat layout and *src* layout. A detailed comparison of the two layouts can be found on the packaging.python.org website.

The flat layout is simpler and more straightforward, the various configuration files and import packages are all in the top-level directory, but it can cause some problems when importing modules or running tests. The *src* layout is more robust and reliable, but it requires some extra configuration and changes in the pyproject.toml/setup.py file. In the *src* layout, the code that is intended to be importable is moved into a subdirectory, e.g.:

```
.
├── README.md
├── noxfile.py
├── pyproject.toml
├── setup.py
├── src/
│   └── awesome_package/
│       ├── __init__.py
│       └── module.py
├── tests/
│   └── test_module.py
└── tools/
    ├── generate_awesomeness.py
    └── decrease_world_suck.py
```

The recommended organisation is the *src* layout.

Once the source code is organised, and the development is completed, the next step is to package the application/package so that it can be installed, distributed, and deployed.
Packaging involves creating a `pyproject.toml` file that contains the metadata and dependencies of the application, as well as other files such as README, LICENSE, MANIFEST, etc., and, if the package provides binary extensions, a `setup.py` to build them. The `pyproject.toml` file allows the application/package to be built and installed using the `setuptools` library, which is the standard tool for packaging Python applications. More information on how to create a `pyproject.toml` file and package a Python application can be found on the packaging.python.org website.
The packaging system as base minimum should be able to produce both source packages (tarballs) and binary packages in "wheel" format (see [PEP427] and updates specifications in packaging.python.org).

2.2. Virtual environments

A virtual environment isolates dependencies required by a software from the rest of the system. The use of virtual environments ensures a stable, reproducible, and portable environment during development and deployment.
The native way for setting up and managing virtual environments is to use the tools included in the Python Standard Library: *venv* and *pip*. *venv* is a small tool that creates and activates/deactivates virtual environments. *pip* is a tool to download and install dependencies (most notably from the Python Package Index, but it can handle other sources as well). These tools are very basic but work well for pure Python dependencies. *pip* can handle non-Python dependencies (e.g. C-libraries such as NumPy): if the package repository has pre-built binaries for the target environment (Python version, operating system, hardware architecture) it will download those, otherwise it will attempt to download the source code and build the package. For the latter (building from source), *pip* assumes that a suitable compiler and libraries are installed, which makes this a fragile workflow.

This problem has been addressed by creating repositories that contain pre-compiled binary packages for all platforms, but notably also provide auxiliary dependencies such as compilers. An auxiliary tool is then used by the developer or end-user to retrieve and install the packages and their dependencies, like *pip*. Among such package managers, *conda* is probably the most well-known and mature one. This tool works with the repositories from Anaconda Inc., a **commercial company that *requires a license to access their repositories***. The *conda* tool itself is open-source, and an alternative, license-free repository is maintained by conda-forge (they also provide *mamba*, an open-source re-implementation of *conda* and fully compatible).

Projects must, as baseline, use *venv* + *pip* to manage virtual environments and dependencies. The required tools for compilation of non-Python sources (e.g. compilers) must be properly documented. Projects may use other tools to simplify the installation process for end users (e.g. *conda*) but their use shall be justified, consistent across the project, and a fallback to manual installation using *venv* + *pip* in an **offline** environment must always be possible.
In addition, vendor lock-in shall be avoided, i.e., do not use *conda* specific features that are not part of the open-source standards. The use of **licenced** repositories and/or packages shall be avoided unless justified and approved.

Many package managers (including *pip* and *conda*) allow to generate “requirements” files or “environment” files including the complete list of dependencies with the corresponding version numbers. Such files can be uses by the same tools that have generated them to setup from the scratch an environment that is identical to the original one. It is good practice to generate and provide such environment files together with the software.

Many package managers (including *pip* and *conda*) ensure the possibility to download all the software dependencies and to generate local package repositories. It is good practice to provide a copy of such local repositories together with the software delivery to allow the user to perform an offline installation.

It is considered bad practice to embed external non-python dependencies (e.g., shared objects for external C/C++ libraries) in binary Python wheel packages. In such cases the use of a

general-purpose package manager (like *conda*) could be considered but considering that it shall be ensured that it remains possible to build wheel packages.

2.3. Writing, running, and debugging Python code

Writing, running, and debugging Python code can be challenging without the use of some tools and best practices. In this section, we will briefly introduce some of the recommended tools and best practices for Python development.

2.3.1. Logging

One of the essential tools for Python development is the **logging package**, or any package that implements its standard interface. Logging is a way of recording events and messages that occur during the execution of a program. Logging can help developers to monitor, troubleshoot, and debug their code, as well as provide useful information for users and administrators. The logging package provides a simple and flexible way of configuring and using logging in Python programs. It also allows developers to use alternative logger implementations by just replacing the import statement.
The software must adhere to the interface of the Python Standard Library's *logging* package. It is recommended to use the *logging* package, but alternative implementations of the interface are allowed. In case that the Software is required to adhere to [E2E-ICD], it needs to implement the log format specified therein. This must instead be achieved using functions provided by the OSFI package or by directly implementing the log format through a custom `logging.Formatter` class.

2.3.2. Type hints and type checking

Type hints are annotations that indicate the expected types of variables, parameters, return values, and attributes in Python code. Type hints were added to Python in version 3.5, and they can help developers to write more readable, maintainable, and robust code. Type hints can also enable static type checking, which is a process of verifying that the types in the code are consistent and correct, before running the program. **Static type checking** can help

developers to catch errors and bugs early, as well as improve the performance and quality of the code. One of the most popular tools for static type checking in Python is MyPy, which can analyse and report type errors in Python code.

However, type hints and static type checking are not enough to ensure the correctness and validity of the data and values in Python code. **Runtime type checking** and data validation are also necessary, especially when dealing with external sources of data, such as user input, files, databases, APIs, etc. Runtime type checking and data validation are processes of verifying that the data and values in the code match the expected types, formats, and constraints, during the execution of the program. Runtime type checking and data validation can help developers to prevent and handle errors and exceptions, as well as ensure the security and integrity of the data. Some of the tools that can perform runtime type checking and data validation in Python are marshmallow, pydantic, typeguard, typical, and pytypes.

2.3.3. Exception handling

Speaking of error and exception handling, Python provides a powerful and flexible mechanism for dealing with unexpected situations and problems that may arise during the execution of a program. Errors and exceptions are objects that represent the occurrence of an event that disrupts the normal flow of the program. Errors and exceptions can be raised by the Python interpreter, the built-in functions and modules, or the user-defined code. Python also provides a way of catching and handling errors and exceptions, using the `try-except-finally` statements. These statements allow developers to specify blocks of code that should be executed when an error or exception occurs, or when the try block is finished**. Error and exception handling must be used by developers** to avoid crashes and terminate the program gracefully, as well as provide useful feedback and recovery options for users and administrators.

2.3.4. Code style

Another aspect of Python development that can improve the readability, maintainability, and consistency of the code is **code style**. Code style refers to the conventions and rules that govern the formatting and structure of the code, such as indentation, whitespace, naming,

comments, etc. Code style can affect the readability and understandability of the code, as well as the collaboration and communication among developers. Python has a set of official guidelines for code style, called [PEP8], which can be found at peps.python.org.
However, following and enforcing code style can be tedious and time-consuming, especially when working on large and complex projects. Therefore, it is advisable to use tools that can automate and simplify the process of checking and formatting code style, such as flake8, black, or ruff.

2.3.5. Documenting code

One more aspect of Python development that can enhance the readability, maintainability, and usability of the code is **documenting code**. Documenting code refers to the practice of adding explanatory and descriptive text to the code, such as comments, docstrings, and external documents. Documenting code can help developers to understand, explain, and maintain their code, as well as provide useful information and instructions for users and other developers who want to use or modify the code. Python has a standard way of writing docstrings, which are multi-line comments that document the purpose, parameters, return values, and behaviour of functions, classes, modules, and packages. The basic standard for docstrings formatting is defined in [PEP257], which can be found at peps.python.org. More advanced docstring formatting standards are the NumPy style docstring format (recommended), the google style docstring format and reStructuredText with Sphinx extensions.
However, writing docstrings alone is not enough to produce comprehensive and accessible documentation for the code. It is also recommended to use tools that can generate documentation from docstrings, such as Sphinx, which can convert docstrings into HTML and CSS files that can be easily viewed and navigated by web browsers. All the above mentioned advanced docstring formatting standards are fully supported by tools like Sphinx.

2.3.6. Object-oriented programming

Finally, one of the paradigms that Python supports and encourages is object-oriented programming. Object-oriented programming is a way of designing and organizing code based on the concepts of objects, classes, inheritance, polymorphism, and encapsulation. Objects are instances of classes, which are templates that define the attributes and methods of the

objects. Inheritance is a mechanism that allows classes to inherit the attributes and methods of other classes and extend or override them. Polymorphism is a feature that allows objects of different classes to be treated uniformly, based on their common interface. Encapsulation is a principle that “hides” the internal details and implementation of the objects and exposes only the relevant and essential information and functionality to the outside world. Object-oriented programming can help developers to **write more modular, reusable, and maintainable code**, as well as **model complex and realistic phenomena and systems**.

Python provides various features and tools that support and facilitate object-oriented programming, such as built-in types, special methods, multiple inheritance, abstract base classes, descriptors, decorators, metaclasses and protocols.

It is strongly recommended that medium to big software project makes use of the object-oriented programming paradigm.

2.4. Test-driven development

One of the best practices for Python development is to adopt a **test-driven approach (TDD)**, where tests are written before the code and used to guide the design and implementation. Test-driven development can improve the quality, reliability, and maintainability of the code, as well as facilitate refactoring and debugging.
The workflow of TDD in Python can be summarized as follows:

- Write a test for a specific feature or functionality and run it. The test should fail since the code for that feature or functionality has not been written yet.
- Write the minimum amount of code that can make the test pass and run it again. The test should pass, indicating that the code meets the requirements of the test.
- Refactor the code to improve its readability, performance, or structure, and run the test again. The test should still pass, ensuring that the code does not break or introduce new errors.
- Repeat the process for each feature or functionality, until the code is complete and all the tests pass.

Using this workflow, TDD can help with developing high-quality Python code that is easy to test, debug, and maintain.

There are various tools and frameworks that can help with TDD in Python, but one of the most popular and widely used is **Pytest**. Pytest is a powerful and flexible testing framework that supports multiple types of tests, such as unit, integration, and end-to-end tests. Pytest also has many features and plugins that enhance its functionality, such as fixtures, parametrization, mocking, parallelization, and integration with other tools.

Another option besides Pytest, is nose2. Nose2 is a successor of nose, a testing framework that extends the built-in unittest module in Python. Nose2 aims to improve the readability and structure of the tests by providing features such as test discovery, plugins, and configuration files. Nose2 also supports Pytest-style fixtures and parametrized tests, which can help with writing more concise and reusable tests. To use nose2 with TDD, developers can follow the same steps as with Pytest but replace Pytest with nose2 in the commands.

In addition to writing tests, it is also important to measure the coverage and effectiveness of the tests. **Code coverage** is a metric that indicates how much of the source code is executed by the tests.
However, code coverage alone does not guarantee that the tests are thorough and meaningful. **Mutation testing** is a technique that can evaluate the quality of the tests by introducing small changes or mutations in the code and checking if the tests can detect them. Mutation testing can be performed in Python using tools such as mutmut or mutpy.

Another way to improve the quality and correctness of the code is to use **type checking**. Type checking is the process of verifying that the types of the variables, arguments, and return values are consistent and compatible. Python is a dynamically typed language, which means that the types are inferred at runtime and not checked beforehand. This can lead to errors and bugs that are hard to detect and debug. To overcome this limitation, Python supports optional type annotations that can be used to specify the expected types of the code elements. Type checking has been introduced in section 2.3.2.

2.5. Configuration management

Configuration management is the practice of keeping track of the changes made to the code and data in a software project. It helps ensure that the code is consistent, reliable, and reproducible across different environments and stages of development. One of the most widely used tools for configuration management is Git, a distributed version control system that allows developers to store, share, and collaborate on code. Git can be integrated with online platforms such as GitHub or GitLab, which provide additional features such as issue tracking, code review, and continuous integration (CI). A CI pipeline is a workflow that automates the process of testing and deploying the code whenever a change is made. By using Git and a CI pipeline, Python developers can improve the quality and efficiency of their software projects.

2.6. Deployment

Depending on the intended use and audience, applications can be distributed and deployed as a package, a module, a script, or an executable file.

2.6.1. Python packages

Python-only projects must be distributed as Python package, using the packaging guidelines published by the Python Packaging Authority. If the project depends on non-Python libraries, then their inclusion shall be compatible with *pip* (via setuptools) and buildable from source.

2.6.2. Dependencies

Dependencies must be version-pinned.
If dependencies from non-standard (i.e., other than PyPI, conda) repositories are used, these shall be provided as separate source packages and their use justified in the Software Reuse File.

2.6.3. Containers

As the size and complexity of the application, a containerization solution might be suitable for deploying Python applications. Container images are self-contained and include all the

dependencies and configurations needed to run the application. This means that the application can run on any environment that supports a container runtime, regardless of the operating system, hardware, or network settings.

The deployment via container, if provided by the developer, is to be understood as optional and complementary to the mandatory deployment directly on the target operating system. In this case, an OCI compliant image format and container runtime shall be used. Vendor lock-in is not permitted (e.g. Docker-specific features).

The developer shall deliver all original sources and build scripts to reproduce the container image from scratch (i.e., a container image is never acceptable as sole deliverable).

3. QUALITY AND PERFORMANCE

3.1. Code quality

Code quality is an essential aspect of the python development process, as it ensures that the code is readable, maintainable, and adheres to the best practices and standards. One way to ensure code quality is to use tools that can perform automated checks and analyses on the code and identify potential issues or improvements.

3.1.1. Formatting

Consistent code formatting guidelines ensure that code remains readable and maintainable, even when multiple developers contribute to the code base. Projects must adopt a code formatting guideline (either an existing standard or tailored variant) and **must** enforce the formatting rules using tools integrated in the CI process. Examples of such tools include black, ruff, flake8, pylint, etc.

3.1.2. Quality metrics

The software development community has developed several metrics that aim to measure quality and maintainability of code. Such metrics include:

- Number of lines of code and comments
- Number of duplicated lines of code

- Comment/code ratio
- Cyclomatic complexity (or McCabe complexity)
- Halstead metrics
- Maintainability index (combining the above metrics into one number)

Projects must, at a minimum, compute the following quality metrics and enforce compliance to the given upper bounds by means of automated tools:

- Lines of code per file <= 1000
- Lines of code per method/function <= 100
- Cyclomatic complexity <= 10
- Maintainability index >= 50

Some tools that can help in computing these metrics include Radon, *pylint* and *flake8*. A comprehensive list of tools for code quality can be found on Python Code Quality Authority website. Note that not all tools use the same definitions for each metric (for example, there exist multiple formula's for computing the maintainability index).

3.1.3. Security

Security and vulnerability scanners are another type of tools that can improve code quality by detecting and preventing security risks and flaws in the code. Projects that are deployed on servers with (semi-) public internet access must perform automated vulnerability checks on the source code **and** all dependencies.

Bandit is a tool that can scan the code for common security issues, such as injection attacks, weak cryptography, hardcoded passwords, and insecure use of subprocesses. Safety is a tool that can check the dependencies of the project and alert the developers if any of them have known vulnerabilities. Both bandit and safety can be run from the command line or as part of a testing framework.

3.2. Performance optimisation

3.2.1. Code Syntax

When optimizing Python code for numerical computations, it is crucial to prioritize efficiency and performance. Here are some refined recommendations:

- **Minimize explicit loops**: developers must utilize Python's powerful abstractions to avoid explicit loops wherever possible. This not only makes code more readable but also leverages Python's internal optimizations for better performance.
- **Leverage iterators**: Developers must opt for iterators or generator expressions over explicit indexing to make code more Pythonic and efficient, especially when dealing with large datasets.
- **Utilize extension packages**: Python's ecosystem is rich with extension packages that can perform tasks more efficiently than custom code. For XML parsing, array operations, and other specialized tasks, rely on these packages to avoid reinventing the wheel and to benefit from their optimized performance.
- **Proper use of NumPy**: For numerical computations, NumPy is the go-to library. It offers C/C++ speed due to its underlying implementations and can significantly speed up operations with its support for SIMD (Single Instruction, Multiple Data) and releasing the GIL (Global Interpreter Lock). Ensure you're using NumPy effectively to tap into these performance benefits.

Using the same numerical library for all the software modules in a project can help achieve overall consistency.

3.2.2. Just-in-time Compilation of Python Code

It is also beneficial to consider just-in-time (JIT) compilation solutions to further enhance code performance. **Numba** and **Taichi** are two notable JIT compilers that can significantly accelerate Python code execution, especially for numerical computations.

- **Numba**: A JIT compiler that translates a subset of Python and NumPy code into fast machine code. Numba is particularly useful for functions that perform heavy numerical computations. It works by decorating Python functions with `@jit` to indicate they should be optimized. Numba then compiles these functions at runtime, resulting in performance that can approach that of C or Fortran.

- **Taichi**: An open-source computer graphics library that provides a Pythonic interface for writing highly parallel computations. Taichi is designed to simplify the development process by abstracting away the complexities of parallel computing, while also offering JIT compilation for performance optimization.

Both Numba and Taichi can be seamlessly integrated into Python workflows, allowing developers to maintain the readability and simplicity of Python code while benefiting from the performance improvements of JIT compilation. These tools are particularly advantageous when dealing with large-scale numerical simulations or data processing tasks that require high computational efficiency

However, using JIT technologies like Numba and Taichi also has some drawbacks, such as:

- They are complex dependencies that may be difficult to install and manage in different environments.
- They usually support only a subset of Python and NumPy features and syntax and may not be compatible with the latest versions or other libraries.
- They require modifying the Python code to use specific decorators, data types, or functions, which can reduce the readability, expressiveness, and idiomaticity of the code.
- They may introduce unexpected errors or bugs due to the runtime compilation or the differences between the Python and the native code.

Therefore, developers should weigh the pros and cons of using JIT technologies and consider their specific use cases and requirements before adopting them. Alternatively, they can explore other options for improving the performance of Python code, such as Cython, PyPy, or C extensions.

3.2.3. Just-in-Time compilation of low-level code

For just-in-time compilation of low-level code, several tools are available:

- **PyOpenCL** offers Pythonic access to the OpenCL parallel computation API, allowing the construction of OpenCL kernels and buffers, complete with support for a NumPy-like array type.

- **PyCUDA** provides a straightforward interface to Nvidia's CUDA parallel computation API, enabling the building of CUDA kernels and buffers, also supporting a numpy-like array type.
- **cppyy** stands as an automatic, run-time Python-C++ bindings generator, facilitating the calling of C++ from Python and vice versa. It boasts features like run-time generation for higher performance, lazy loading, Python-side cross-inheritance, and callbacks for C++ frameworks, among others.
- **xobjects** offers in-memory serialization of structured types with C-API generation and compiles run-time C code using cffi, cupy, and pyopencl under a unified API.

These tools collectively empower developers to optimize Python code by leveraging the capabilities of low-level programming, ensuring that applications run at the highest possible efficiency.

Compile Python modules and extensions

- Cython: Cython is an optimising static compiler for both the Python programming language and the extended Cython programming language. It makes writing C/C++ extensions for Python as easy as Python itself. Cython allows to access and use C/C++ libraries easily and hence it is also used to generate wrappers/bindings for libraries written in such languages.
- Pythran: Pythran is an ahead of time compiler for a subset of the Python language, with a focus on scientific computing. It takes a Python module annotated with a few interface descriptions and turns it into a native Python module with the same interface, but (hopefully) faster. It is meant to efficiently compile scientific programs and takes advantage of multi-cores and SIMD instruction units.
- Mypyc: Mypyc compiles Python modules to C extensions. It uses standard Python type hints to generate fast code.
- Nuitka: Nuitka is the optimizing Python compiler written in Python that creates executables that run without a need for a separate installer. Data files can both be included or put alongside.
- Codon: Codon is a high-performance Python compiler that compiles Python code to native machine code without any runtime overhead.

Bind Low-Level Code Modules to Python

A keyway to boost performance in Python development is to connect Python to modules that use low-level code. Here are some tools that can assist with this task:

- **Pybind11** is a lightweight, header-only library that exposes C++ types in Python and vice versa, primarily to create Python bindings of existing C++ code.
- **Cython** is an optimising static compiler for both the Python programming language and the extended Cython programming language. It makes writing C extensions for Python as easy as Python itself. Cython allows to access and use C/C++ libraries easily and hence it is also used to generate wrappers/bindings for libraries written in such languages.
- **Nanobind** offers similar functionality to Pybind11 and Boost.Python but with quicker compilation times, smaller binaries, and improved runtime performance.
- **Maturin** is a build system for Rust-based Python packages, utilizing PyO3 for seamless bindings.
- **Nimporter** compiles Nim extensions for Python on import, leveraging nimpy for on-the-fly compilation.
- **SWIG** connects C and C++ programs with various high-level languages, including Python.
- **ctypes** is a foreign function library in Python that allows calling functions in DLLs or shared libraries.
- **cffi** provides the ability to interact with almost any C code from Python, based on C-like declarations. These tools collectively enable developers to maintain the high-level expressiveness of Python while harnessing the power of low-level languages, ensuring that applications run with optimal efficiency.

It is strongly recommended to use the same method for creating bindings from Python to other languages for each set of software modules in a project.

4. RECOMMENDED PACKAGES

4.1. Packages based on Python array types

Many packages have been developed to enhance the functionality and performance of Python arrays, offering various features and benefits for different use cases. Here are some of the most recommended packages based on Python array types:

- Numpy: Numpy is the fundamental package for scientific computing with Python. It provides N-dimensional arrays with comprehensive vectorized operations: mathematical functions, random number generators, linear algebra routines, Fourier transforms written in low-level C code.
- Xarray: Xarray introduces labels in the form of dimensions, coordinates and attributes on top of raw NumPy-like multidimensional arrays, which allows for a more intuitive, more concise, and less error-prone developer experience.
- Dask: Dask is a flexible library for parallel computing in Python. It provides dynamic task scheduling and out-of-memory big data collections.
- JAX: JAX uses an improved Autograd implementation in combination with XLA to compile and run your Python/NumPy programs on CPUs, GPUs and TPUs. It enables composable function transformations and can differentiate through loops, branches, recursion, closures, and it can take derivatives of derivatives of derivatives.
- PyTorch: PyTorch is a tensor computation (like NumPy) library with strong GPU acceleration that enables building deep neural networks on a tape-based autograd system. It includes data structures for multi-dimensional tensors and defines mathematical operations over these tensors, as well as utilities for efficient serializing of Tensors and arbitrary types, efficient compiling of ML models, and other useful utilities.
- CuPy: CuPy is a NumPy/SciPy-compatible array library for GPU-accelerated computing with Python. It acts as a drop-in replacement to run existing NumPy/SciPy code on NVIDIA CUDA or AMD ROCm platforms. It is essentially NumPy & SciPy for GPU. You can also easily make a custom CUDA kernel if you want to make your code run faster, requiring only a small code snippet of C++.

A consistent way of dealing with array types should be used throughout the project's software components, e.g. using both Xarray arrays and NumPy arrays should be discouraged.

Required dependencies shall be kept to a minimum.

4.2. Fast DataFrame-type computations

For those who need to work with large-scale tabular data in Python, some packages that offer DataFrame-type computations on GPU or out-of-core memory may be of interest. These packages can manage massive datasets that exceed the available RAM and offer high-performance operations such as filtering, grouping, aggregating, joining, and more. Here are some of the most recommended packages

- Pandas: is a fast, powerful, flexible and easy to use open-source data analysis and manipulation tool.
- Polars: Polars is a lightweight, fast multi-threaded, hybrid streaming DataFrame library written in Rust using the Apache Arrow columnar format. It enables fast out-of-memory operations, lazy/eager execution, query optimization and more.
- cuDF: cuDF is a Python GPU DataFrame library built on the Apache Arrow columnar memory format with a pandas-like API.
- Vaex: Vaex is a highly performant library for lazy out-of-core DataFrames, to visualize and explore big tabular datasets. It can apply operations on an N-dimensional grid up to a billion (109) objects/rows per second and provides a set of sub-packages for various applications (visualisation, jupyter integration, data formats support, machine learning etc.)
- PySpark: PySpark is an interface for Apache Spark in Python, with support for most of Spark's features such as Spark SQL, DataFrame, Streaming, MLlib (Machine Learning) and Spark Core.
- Modin: Modin is a drop-in replacement for pandas to instantly speed up your workflows by scaling pandas, so it uses all your cores. It is most likely the slowest barrier to entry for performance improvements on DataFrame operations: changing the import line is enough.

4.3. Geo-spatial and satellite data processing tools

Python has many tools for working with geospatial data in raster or vector formats, and for doing some of the usual steps in satellite data processing. This section gives a summary of the available tools:

- SentiNel Application Program (SNAP): SNAP is a software platform developed by ESA. It consists of multiple toolboxes that can be used for image processing, modelling and visualization. Although the toolboxes are oriented towards Sentinel missions, they can be used with other Earth observation missions. SNAP is accessible from python through the **snappy** package.
- Orfeo Toolbox (OTB): OTB is a software library for processing images from Earth observation satellites accessible from python through an API. It is an open-source development, and it can process high resolution optical, multispectral and radar images at the terabyte scale. A wide variety of applications are available including, among others, ortho-rectification, pansharpening, classification and SAR processing.
- Geospatial Data Abstraction Library (GDAL): GDAL is a translator library for raster and vector geospatial data formats that is released under an MIT style Open-Source License. It handles raster and vector formats and can apply a variety of data translation and processing algorithms. Projections and transformations are supported by the PROJ library. Tools for programming and manipulating the GDAL are available as a Python package.
- Rasterio: Geographic information systems use GeoTIFF and other formats to organize and store gridded raster datasets such as satellite imagery and terrain models. Rasterio reads and writes these formats and provides a Python API based on Numpy N-dimensional arrays and GeoJSON.
- Fiona: Fiona streams simple feature data to and from GIS formats like GeoPackage and Shapefile. Fiona can read and write using multi-layered GIS formats, zipped and in-memory virtual file systems. This project includes Python modules and a command line interface (CLI).

5. CONCLUSIONS

In this document, we have introduced some of the guidelines and best practices for Python development. These are not mandatory rules, but rather recommendations that can help Python developers write code that is consistent, readable, maintainable, and adheres to the principles and idioms of the Python language.

By following these guidelines and best practices, Python developers can improve the quality of their code, avoid common errors and pitfalls, and make their code easier to understand, reuse, and collaborate with others. Moreover, they can benefit from the tools and frameworks that support and enforce these standards, such as code formatters, linters, testing frameworks, and code editors.

Python is a powerful and expressive language that offers many features and possibilities for various applications and domains. However, with great power comes great responsibility, and Python developers should strive to use the language in a way that respects its philosophy and enhances its beauty and elegance.