



Python Code Quality: Best Practices and Tools

by [Leodanis Pozo Ramos](#)

🕒 Mar 24, 2025

💬 8 Comments



intermediate

best-practices

python

to

Mark as Completed



Table of Contents


- [Defining Code Quality](#)
 - [Low-Quality Code](#)
 - [High-Quality Code](#)
 - [The Importance of Code Quality](#)
- [Exploring Code Quality in Python With Examples](#)
 - [Functionality](#)
 - [Readability](#)
 - [Documentation](#)
 - [Compliance](#)
 - [Reusability](#)
 - [Maintainability](#)
 - [Robustness](#)
 - [Testability](#)
 - [Efficiency](#)
 - [Scalability](#)
 - [Security](#)
- [Managing Trade-Offs Between Code Quality Characteristics](#)
- [Applying Best Practices for High-Quality Code in Python](#)
 - [Style Guides](#)
 - [Code Reviews](#)
 - [AI Assistants](#)
 - [Documentation Tools](#)
 - [Code Linters](#)
 - [Static Type Checkers](#)
 - [Code Formatters](#)

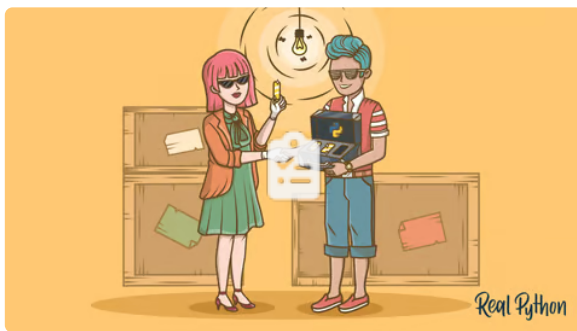
By the end of this tutorial, you'll understand that:

- **Checking the quality of Python code** involves using tools like linters and static type checkers to enforce coding standards and detect potential errors.
- **Writing quality code in Python** requires following best practices, such as clear naming conventions and comprehensive testing.
- **Good Python code** is characterized by readability, maintainability, efficiency, and adherence to community standards.
- **Making Python code look good** involves using formatters to ensure consistent styling and readability according to established coding styles.
- **Making Python code readable** means using descriptive names for variables, functions, classes, and modules.

Read on to learn more about the strategies, tools, and best practices that will help you write high-quality Python code.

Get Your Code: [Click here to download the free sample code](#) that you'll use to learn about Python code quality practices and tools.

 **Take the Quiz:** Test your knowledge with our interactive “Python Code Quality: Best Practices and Tools” quiz. You'll receive a score upon completion to help you track your learning progress:



Interactive Quiz

[Python Code Quality: Best Practices and Tools](#)

In this quiz, you'll test your understanding of Python code quality. As you're working through this quiz, you'll revisit the importance of producing code that's functional, readable, maintainable, efficient, and secure.

Defining Code Quality

Of course you want quality code. Who wouldn't? But what is **code quality**? It turns out that the term means different things to different people.

One way to approach code quality is to look at the two ends of the quality spectrum:

- **Low-quality code:** It has the **minimal required** characteristics to be **functional**.
- **High-quality code:** It has **all the necessary** characteristics that make it work reliably, efficiently, and securely.

may stop using it altogether.

While simplistic, these two characteristics are generally accepted as the baseline of functional but low-quality code. The code may work, but it often lacks readability, maintainability, and efficiency, making it difficult to scale.

High-Quality Code

Now, here's an extended list of the key characteristics that define high-quality code:

- **Functionality:** Works as expected and fulfills its intended purpose.
- **Readability:** Is easy for humans to understand.
- **Documentation:** Clearly explains its purpose and usage.
- **Standards Compliance:** Adheres to conventions and guidelines, such as PEP 8.
- **Reusability:** Can be used in different contexts without modification.
- **Maintainability:** Allows for modifications and extensions without introducing bugs.
- **Robustness:** Handles errors and unexpected inputs effectively.
- **Testability:** Can be easily verified for correctness.
- **Efficiency:** Optimizes time and resource usage.
- **Scalability:** Handles increased data loads or complexity without degradation.
- **Security:** Protects against vulnerabilities and malicious inputs.

In short, **high-quality code** is functional, readable, maintainable, and robust. It follows best practices, has a consistent coding style, modular design, proper error handling, and adherence to coding standards. It is also easy to test and scale. Finally, high-quality code is efficient and secure, ensuring reliability and safety.

All the characteristics above allow developers to understand, modify, and extend a Python codebase.

The Importance of Code Quality

To understand why code quality matters, you'll revisit the characteristics of high-quality code from the previous section and examine their impact:

- **Functional code:** Ensures correct behavior and expected outcomes.
- **Readable code:** Makes understanding and maintaining code easier.
- **Documented code:** Clarifies the correct and recommended way for others to use it.
- **Compliant code:** Promotes consistency and allows collaboration.
- **Reusable code:** Saves time by allowing code reuse.

The most important factor when evaluating the quality of a piece of code is whether it can do what it was intended to do. If this factor isn't achieved, then there's no room for discussion about the code's quality.

Consider the following quick example of a [function](#) that adds two numbers. You'll start with a low-quality function.

Low-quality code:

Python

```
>>> def add_numbers(a, b):  
...     return a + b  
...  
  
>>> add_numbers(2, 3)  
5
```

Your `add_numbers()` function seems to work well. However, if you dig deeper into the implementation of some argument types, then the function will crash:

Python

```
>>> add_numbers(2, "3")  
Traceback (most recent call last):  
...  
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

In this call to `add_numbers()`, you pass an integer and a string. The function tries to add them, but Python crashes because it's impossible to add numbers and strings. Now, it's time for a higher-quality implementation.

Higher-quality code:

Python

```
>>> def add_numbers(a: int | float, b: int | float) -> float:  
...     a, b = float(a), float(b)  
...     return a + b  
...  
  
>>> add_numbers(2, 3)  
5.0  
  
>>> add_numbers(2, "3")  
5.0
```

The following example shows why readability is important. Again, you'll first have a low-quality version.

Low-quality code:

Python

```
>>> def ca(w, h):  
...     return w * h  
...  
  
>>> ca(12, 20)  
240
```

This function works. It takes two numbers and multiplies them, returning the result. But can you tell? Consider the improved version below.

Higher-quality code:

Python

```
>>> def calculate_rectangle_area(width: float, height: float) -> float:  
...     return width * height  
...  
  
>>> calculate_rectangle_area(12, 20)  
240
```

Now, when you read the function's name, you immediately know what the function is about because it provides additional context.

Documentation

Documenting code is a task that gets little love among software developers. However, clear and well-documented code is essential for evaluating the quality of any software project. Below is an example of how documentation improves code quality.

Low-quality code:

Python

```

...         float: Product of a and b.
...         """
...         return a * b
...

>>> multiply(2, 3)
6

```

In the function's docstring, you provide context that lets others know what the function does and what it takes. You also specify its return value and corresponding data type.

Compliance

Meeting the requirements of well-known and widely accepted code standards is another key factor to writing a piece of code. The relevant standards will vary depending on the project at hand. A good generic example is code that follows the standards and conventions established in [PEP 8](#), the official style guide for Python. It's a good idea to avoid low-quality code that doesn't follow PEP 8 guidelines.

Low-quality code:

Python

```

>>> def calcTotal(price,taxRate=0.05): return price*(1+taxRate)
...

>>> calcTotal(1.99)
2.0895

```

This function doesn't follow the naming conventions and spacing norms established in [PEP 8](#). The code doesn't look like quality Python code. It isn't Pythonic. Now for the improved version.

Higher-quality code:

Python

```

>>> def calculate_price_with_taxes(
...     base_price: float, tax_rate: float = 0.05
... ) -> float:
...     return base_price * (1 + tax_rate)
...

>>> calculate_total_price(1.99)
2.0895

```

```
...
>>> greet_alice()
'Hello, Alice!'
```

This function hardcodes its use case. It only works when you want to greet Alice, which is pretty restrictive. Let's look at an enhanced version below.

✓ Higher-quality code:

Python

```
>>> def greet(name: str) -> str:
...     return f"Hello, {name}!"
...

>>> greet("Alice")
'Hello, Alice!'
>>> greet("John")
'Hello, John!'
>>> greet("Jane")
'Hello, Jane!'
```

Although quite basic, this function is more generic and useful than the previous version. It takes a parameter and builds a greeting message using an [f-string](#). Now, you can greet all your friends!

Maintainability

Maintainability is all about writing code that you or other people can quickly understand, update, extend, etc. Reducing repetitive code and code with multiple responsibilities are key principles to achieving this quality characteristic. Let's look at the example below.

● Low-quality code:

Python

```
>>> def process(numbers):
...     cleaned = [number for number in numbers if number >= 0]
...     return sum(cleaned)
...

>>> print(process([1, 2, 3, -1, -2, -3]))
6
```

Robustness

Writing robust code is also fundamental in Python or any other language. Robust code is capable of preventing crashes and unexpected behaviors and results. Check out the example below, where you divide two numbers.

Low-quality code:

Python

```
>>> def divide_numbers(a, b):
...     return a / b
...

>>> divide_numbers(4, 2)
2.0
>>> divide_numbers(4, 0)
Traceback (most recent call last):
...
ZeroDivisionError: division by zero
```

This function divides two numbers, as expected. However, when the divisor is 0, the code breaks with an exception. To fix the issue, you need to properly handle the exception.

Higher-quality code:

Python

```
>>> def divide_numbers(a: float, b: float) -> float | None:
...     try:
...         return a / b
...     except ZeroDivisionError:
...         print("Error: can't divide by zero")
...

>>> divide_numbers(4, 2)
2.0
>>> divide_numbers(4, 0)
Error: can't divide by zero
```

Now, your function handles the exception, preventing a code crash. Instead, you print an informative

Python

```
import pytest

def test_greet(capsys):
    greet("Alice")
    captured = capsys.readouterr()
    assert captured.out.strip() == "Hello, Alice!"
```

This test case works. However, it's hard to write because it demands a relatively advanced knowledge

You can replace the call to `print()` with a `return` statement to improve the testability of `greet()` and

✓ Higher-quality code:

Python

```
def greet(name: str) -> str:
    return f"Hello, {name}!"

def test_greet():
    assert greet("Alice") == "Hello, Alice!"
```

Now, the function returns the greeting message. This makes the test case quicker to write and require less code. It's also more efficient and quick to run, so this version of `greet()` is more testable.

Efficiency

Efficiency is another essential factor to take into account when you have to evaluate the quality of a program. You can think of efficiency in terms of **execution speed** and **memory consumption**.

Depending on your project, you may find other features that could be considered for evaluating efficiency, such as network latency, energy consumption, and many others.

Consider the following code that computes the [Fibonacci](#) sequence of a series of numbers using a recursive function.

🔴 Low-quality code:

Python

efficiency_v1.py

✓ Higher-quality code:

Python

efficiency_v2.py

```
from time import perf_counter

cache = {0: 0, 1: 1}

def fibonacci_of(n):
    if n in cache:
        return cache[n]
    cache[n] = fibonacci_of(n - 1) + fibonacci_of(n - 2)
    return cache[n]

start = perf_counter()
[fibonacci_of(n) for n in range(35)] # Generate 35 Fibonacci numbers
end = perf_counter()

print(f"Execution time: {end - start:.2f} seconds")
```

This implementation optimizes the Fibonacci computation using caching. Now, run this improved code again:

Shell

```
$ python efficiency_v1.py
Execution time: 0.01 seconds
```


Wow! This code is quite a bit faster than its previous version. You've improved the code's efficiency by a factor of 100.

Python Tricks The Book

A Buffet of Awesome Python Features

[Get Your Free Sample Chapter](#)



 [Remove ads](#)

Scalability

The scalability of a piece of code refers to its ability to handle increasing workload, data size, or user count without compromising the code's performance, stability, or maintainability. It's a relatively complex concept, but here's a quick example of code that deals with increasing data size.

● Low-quality code:

```
>>> sum_even_numbers([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
30
```

The generator expression that you use as the argument to `sum()` ensures that only one value is stored while calculating the total.

Security

Secure code prevents security vulnerabilities, protects sensitive data, and defends against potential attacks. Following best practices ensures that systems stay safe, reliable, and resilient against common security threats.

A good example of risky code is when you accept a user's input without validating it, and use this input to perform an action.

Low-quality code:

Python

input_v1.py

```
user_input = "Amount to withdraw? "
amount = int(input(user_input))
available_balance = 1000
print(f"Here are your {amount:.2f}USD")
print(f"Your available balance is {available_balance - amount:.2f}USD")
```

This code uses the built-in `input()` function to grab the user input. The user should provide the amount to withdraw.

Shell

```
$ python input_v1.py
Amount to withdrawn? 300
Here are your 300.00USD
Your available balance is 700.00USD
```

The script takes the input data, simulates a cash withdrawal, and computes the final balance. Now, let's see what happens if the user enters a value greater than the available amount, as shown in the following:

Shell

```
$ python input_v1.py
Amount to withdrawn? 2000
Here are your 2000.00USD
Your available balance is -1000.00USD
```

In this case, the code has an error because the input value is greater than the available amount, and it doesn't handle the error. As a result, the code gives out more money than it should. While this may seem like something that could happen in a real-world scenario, it's not a good practice.

Managing Trade-Offs Between Code Quality Characteristics

While writing your code, you'll often encounter trade-offs between different code quality characteristics. You should aim to find a good balance. Certain decisions prioritize one characteristic at the expense of another, based on goals, constraints, and requirements.

Here are some common conflicts between code quality characteristics:

Conflict	Description
Readability vs Efficiency	Writing highly-optimized code for processing a dataset may make the code harder to read than using a basic loop.
Maintainability vs Performance	Using multiple small functions can introduce function call overhead compared to a more optimized solution appearing in different places.
Testability vs Efficiency	Adding mock dependencies or extensive logging to make code testable might impact runtime performance.
Scalability vs Readability	Distributed systems or parallelized code can be harder to read or understand than a sequential implementation.
Reusability vs Maintainability	Creating highly generic and reusable components might add unnecessary complexity to a project.
Compliance vs Performance	Following PEP 8 and using type hinting may introduce extra verbosity that impacts performance.
Robustness vs Readability	Adding extensive error handling, such as <code>try ... except</code> blocks and logging, might make the codebase harder to read.

To balance these conflicts, you should consider some of the following factors:

- **Project requirements:** What is the primary goal?

Characteristic	Strategy	Resources
Functionality	Adhere to the requirements	Requirement documents
Readability	Use descriptive names for code objects	PEP 8 , code reviews
Documentation	Use docstrings, inline comments, good README files, and external documentation	PEP 257 , README guide
Compliance	Follow relevant standards, use a coding style guide consistently, use linters, formatters, and static type checkers	PEP 8
Reusability	Write parameterized and generic functions	Design principles (SOLID), Refactoring techniques
Maintainability	Write modular code, apply the don't repeat yourself (DRY) and separation of concerns (SoC) principles	Design patterns , code refactoring techniques
Robustness	Do proper error handling and input validation	Exception handling guides
Testability	Write unit tests and use test automation	Code testing guides
Efficiency	Use appropriate algorithms and data structures	Algorithms , Big O notation , data structures
Scalability	Use modular structures	Software architecture topics
Security	Sanitize inputs and use secure defaults and standards	Security best practices


Of course, this table isn't an exhaustive summary of strategies, techniques, and tools for achieving co

case with Google, which has a well-crafted [coding style guide](#) for their Python developers.

Other than that, some Python projects have established their own style guides with guidelines specific to their project. An example is the [Django](#) project, which has a dedicated [coding style guide](#).

Write Cleaner & More Pythonic Code

realpython.com

 [Remove ads](#)



Code Reviews

A **code review** is a process where developers examine and evaluate each other's code to ensure it meets the requirements before merging it into the main or production codebase. The process may be carried out by experienced developers reviewing the code of other team members.

A code reviewer may have some of the following responsibilities:

- Verifying the code works as expected
- Ensuring the code is clear, with meaningful variable names and proper comments
- Catching repetitive code, encouraging modular design, and promoting reusable functions
- Identifying logic errors, edge cases, and incorrect assumptions
- Ensuring the code properly handles errors, edge cases, and exceptional situations
- Confirming the code adheres to style guidelines
- Detecting vulnerabilities like lack of input validation or unsafe usage of functions and objects
- Spotting inefficient algorithms or resource-intensive operations
- Ensuring the code has tests and satisfactory test coverage

As you can conclude, a good code review involves checking for code correctness, readability, maintainability, and other quality factors.

To conduct a code review, a reviewer typically uses a specialized platform like [GitHub](#), which allows for providing feedback in the form of comments, code change suggestions, and more.

Code reviews are vital for producing high-quality Python code. They improve the code being reviewed, help developers learn, and collectively promote coding standards and best practices across the team.

- Use Pythonic approaches and practices
- Generate docstrings, comments, README files, and documentation
- Write unit tests for your code and ensure good test coverage
- Detect common security vulnerabilities and issues
- Identify inefficient algorithms and write faster alternatives

AI assistants and LLMs can also act as interactive mentors for junior developers. To explore how to use AI to improve the quality of your code, you can check out the following resources:

- [Prompt Engineering: A Practical Example](#)
- [ChatGPT: Your Personal Python Coding Mentor](#)
- [Real Python Code Mentor](#)
- [Episode 174: Considering ChatGPT's Technical Review of a Programming Book](#)
- [Episode 236: Simon Willison: Using LLMs for Python Development](#)

You can make it your goal to use AI to improve both your code quality and general development workflow. AI may not replace you, those who learn to use it effectively will have an advantage.

Documentation Tools

To document your Python code, you can take advantage of several available tools. Before taking a look at some tools, it's important to mention [PEP 257](#), which describes conventions for Python's docstrings.

A [docstring](#) is typically a triple-quoted string that you write at the beginning of modules, functions, and classes to provide the most basic layer of [code documentation](#). Modern [code editors and integrated development environments](#) take advantage of docstrings to display context help when you're working on your code.

As a bonus, there are a few handy tools to generate documentation directly from the code by reading docstrings. About a couple of these tools, check out the following resources:

- [Build Your Python Project Documentation With MkDocs](#)
- [Documenting Python Projects With Sphinx and Read the Docs](#)

Probably the most common piece of documentation that you'd write for a Python project is a README file. A README file is what you typically add to the root directory of a software project, and is often a short guide that provides context about the project. To dive deeper into best practices and tools for writing a README, check out the [Creating a README for Your Python Projects](#) tutorial.

You can also take advantage of AI, LLMs, and chatbots to help you generate detailed and high-quality documentation.

function signatures.

- **Error handling issues:** Identify overly broad exceptions and empty except blocks.
- **Undocumented code:** Check for missing docstrings in modules, classes, functions, and so on.
- **Best practices:** Advise against poor coding practices, such as mutable default argument values.
- **Security vulnerabilities:** Warn against insecure coding practices, such as hardcoded passwords, the use of `eval()` and `exec()`.
- **Code duplication:** Alert when similar code blocks appear multiple times.

Here are some modern Python linters with brief descriptions:

Linters	Description
Pylint	A linter that checks for errors, enforces PEP 8 coding style standards, detects code smells, and measures code complexity.
Flake8	A lightweight linter that checks for PEP 8 compliance, syntax errors, and common coding errors, with plugins.
Ruff	A fast, Rust-based tool that provides linting, code formatting, and type checking. It aims to be a replacement for Flake8 and Black.

To run these linters against your code, you use different commands depending on the tool of choice. These tools can be nicely integrated into the majority of Python [code editors and IDEs](#), such as [Visual Studio](#).

These editors and IDEs have the ability to run linters in the background as you type or save your code. They can underline, or otherwise identify problems in the code before you run it. Think of this feature as an automatic linter.

Static Type Checkers

Static type checkers are another category of code checker tools. Unlike linters, which cover a wide range of issues, static type checkers focus on validating type [annotations](#) or type hints. They catch possible type errors before running the code.

Some aspects of your code that you can check with static type checkers include the following:

- **Incorrect type usage in variables and function calls:** Detect variables used with an incompatible type or functions called with arguments of wrong types.
- **Missing or incorrect annotations:** Warn if a piece of code lacks proper type hints.
- **Type inference errors:** Identify places where inferred types don't match expectations.

Code Formatters

A code formatter is a tool that automatically parses your code and formats it to be consistent with the standard. Usually, the standard path is to use PEP 8 as the style reference. Running a code formatter against your code can result in related changes, including the following:

- **Consistent indentation:** Converts all indentation to a standard indentation of four spaces.
- **Line length:** Wraps lines exceeding length limits.
- **Spacing and padding:** Adds or removes spaces for readability, such as around operators and assignment.
- **String quotes consistency:** Converts all strings to a single quote or double quote style.
- **Import sorting:** Ensures imports are alphabetically ordered and grouped logically.
- **Consistent use of blank lines:** Ensures blank lines between functions and classes.

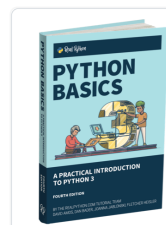
Here are some code formatting tools:

Formatter	Description
Black	Formats Python code without compromise
Isort	Formats imports by sorting alphabetically and separating into sections
Ruff	Provides linting, code formatting, and import sorting

You can install and run these code formatters against your code from the command line. Again, you'll have extensions to incorporate them into your workflow.

Typically, these tools will apply the formatting when you save the changes, but they also have options to format on file open, paste, type, and so on.

If you work on a team of developers, then you can configure everyone's environment to use the same defaults and rely on the automated formatting. This frees you from having to flag formatting issues.



Your **Practical Introduction to Python 3**

 [Remove ads](#)

In Python, you have several tools to help you write and automate tests. You can use `doctest` and `unittest` libraries, or you can use the `pytest` third-party library. To learn about these tools and how to test your code, check out the following resources:

- [Getting Started With Testing in Python](#)
- [Python's `doctest`: Document and Test Your Code at Once](#)
- [Python's `unittest`: Writing Unit Tests for Your Code](#)
- [Effective Python Testing With `pytest`](#)

As a bonus, you can also use an AI assistant to help you write tests for your Python code. These tools can significantly increase your productivity. Check out the [Write Unit Tests for Your Python Code With ChatGPT](#) tutorial for a quick guide.

Code Profilers

Code profilers analyze a program's runtime behavior. They measure performance-related metrics, such as execution time, memory usage, CPU usage, and function call frequencies. A code profiler flags the parts of your code that are inefficient, allowing you to make changes to optimize the code's efficiency.

Running code profiling tools on your code allows you to:

- **Identify performance bottlenecks** like slow functions or loops that impact execution time.
- **Detect intensive CPU usage or high memory consumption**, letting you optimize resource usage.
- **Spot inefficient algorithms**, allowing you to replace inefficient code with optimized solutions.
- **Detect high latency**, letting you speed up operations like database queries, API calls, or computations.
- **Find scalability issues** related to handling larger datasets or higher loads.
- **Make good refactoring decisions** based on detailed data and statistics.

Common Python profiling tools include the following:

Tool	Description
cProfile	Measures function execution times
timeit	Times small code snippets
perf	Finds hot spots in code, libraries, and even the operating system's code
py-spy	Finds what a program is spending time on without restarting it or modifying its code


[Bandit](#) is a security-focused linter that scans for common vulnerabilities and insecure coding patterns. These patterns include the use of:

- Unsanitized user inputs
- The `exec()`, `eval()`, and `pickle.load()` functions
- The `assert` statement
- Hardcoded API keys or passwords
- Insecure file operations
- Shell commands
- Unencrypted HTTP requests instead of HTTPS
- Broad exceptions
- Hardcoded SQL queries

These are just a sample of the tests that Bandit can run against your code. You can even use it to [write](#) tests to run them on your Python projects. Once you have a list of detected issues, you can take action and fix them to make your code more secure.

A Python Best Practices Handbook

python-guide.org

 [Remove ads](#)

Deciding When to Check the Quality of Your Code

You should check the quality of your code frequently. If automation and consistency aren't there, it's easy for a project to lose sight of the goal and start writing lower-quality code. It can happen slowly, one tiny issue at a time. Over time, all those issues pile up and eventually, you can end up with a codebase that's buggy, hard to maintain, and a pain to extend with new features.

To avoid falling into the low-quality spiral, check your code often! For example, you can check the code quality by:

- Write it
- Commit it
- Test it

You can use code linters, static type checkers, formatters, and vulnerability checkers as you write code.

checkers, and formatters. You also delved into strategies and techniques such as code reviews, testing

Writing high-quality code is crucial for you as a Python developer. High-quality code reduces development errors, and facilitates collaboration between coworkers.

In this tutorial, you've learned how to:

- Identify **characteristics** of **low** and **high-quality** code
- Implement **best practices** to enhance code readability, maintainability, and efficiency
- Use tools like **linters**, **type checkers**, and **formatters** to enforce code quality
- Use effective **code reviews** and **AI assistance** to achieve code quality
- Apply **testing** and **profiling** techniques to ensure code robustness and performance

With this knowledge, you can now confidently write, review, and maintain high-quality Python code, an efficient development process and robust applications.

Get Your Code: [Click here to download the free sample code](#) that you'll use to learn about Python practices and tools.

Frequently Asked Questions

Now that you have some experience with enhancing Python code quality, you can use the questions to test your understanding and recap what you've learned.

These FAQs are related to the most important concepts you've covered in this tutorial. Click the *Show* question to reveal the answer.

How do I check the quality of Python code?

What are the key characteristics of high-quality Python code?

How do you write high-quality Python code?

What tools can I use to improve Python code quality?

days. No spam ever. Unsubscribe any time. Curated by the Real Python team.

```
2 # ...  
3  
4 >>> x =  
5 >>> y =  
6  
7 >>> z =  
8  
9 >>> z  
10 {'c': 4,
```

Email Address



Send Me Python Tricks »

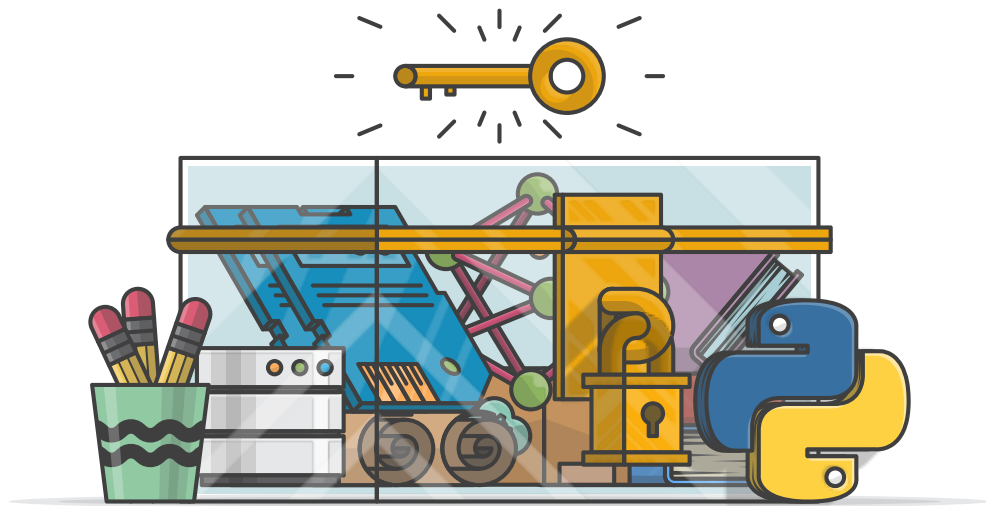
About **Leodanis Pozo Ramos**



Leodanis is an industrial engineer who loves Python and software development. He's a self-taught experience. He's an avid technical writer with a growing number of articles published on Real Python.

» [More aboutLeodanis](#)

Master Real-World Python Skills With Unlimited Access



Join us and get access to thousands of tutorials, hands-on video courses, and a community of expert Pythonistas:

[Level Up Your Python Skills »](#)

What Do You Think?

Rate this article:



[LinkedIn](#)

[Twitter](#)

[Bluesky](#)

[Facebook](#)


[Email](#)

- [How to Write Beautiful Python Code With PEP 8](#)
- [Getting Started With Testing in Python](#)
- [Variables in Python: Usage and Best Practices](#)
- [Python Type Checking \(Guide\)](#)

Your Guide to the Python Programming Language and a Best Practices Handbook

python-guide.org



 [Remove ads](#)

© 2012–2025 Real Python • [Newsletter](#) • [Podcast](#) • [YouTube](#) • [Twitter](#) • [Facebook](#) • [Instagram](#) • [Python Tutorials](#) • [Search](#) • [Privacy Policy](#) • [Energy Policy](#) • [Advertise](#)

♥ Happy Pythoning!