Code Formatting and Linting in Python

What Are Code Formatting and Linting?

As Python projects grow, code can become difficult to read and maintain.

Two important development practices help keep Python code clean:

1. **Code formatting**
2. **Linting**

They solve different problems.

Code formatting

Formatting focuses on **how code looks**.

For example:

```
def calculate_total(price,quantity):
    return price*quantity
```

A formatter can turn it into:

```
def calculate_total(price, quantity):
    return price * quantity
```

Linting

Linting focuses on **potential problems and code quality**.

For example:

```
unused_value = 10
```

A linter may warn that `unused_value` is never used.

Think of it this way:

```
Formatting
    ↓
How should the code look?

Linting
    ↓
Is there something suspicious or problematic?
```

Why Formatting and Linting Matter

A project can work correctly while still having messy or inconsistent code.

Consider:

```
def get_user(id):
 x=10
 if x>5:
  print("User found")
```

The code may technically run, but it is harder to read.

Formatted code:

```
def get_user(user_id):
    x = 10

    if x > 5:
        print("User found")
```

Now the structure is easier to understand.

In a real project, formatting and linting help with:

- readability
- consistency
- maintainability
- code review
- catching common mistakes
- reducing unnecessary discussions about style
- cleaner pull requests
- faster onboarding
- automated quality checks

Formatting vs Linting

These concepts are often confused.

Practice	**Main Question**
Formatting	How should the code be

	written visually?
Linting	Is the code suspicious, incorrect, or unnecessarily problematic?
Type checking	Are values being used according to their declared types?
Testing	Does the program behave correctly?

A modern Python project may use all four.

```
Formatter
    ↓
Consistent appearance

Linter
    ↓
Code quality

Type checker
    ↓
Type correctness
```

```
Tests
    ↓
Behavior correctness
```

PEP 8

Python has a widely followed style guide called **PEP 8**.

PEP 8 provides recommendations for writing readable Python code.

It covers things such as:

- indentation
- line length
- whitespace
- naming
- imports
- blank lines
- expressions
- comments
- code layout

Example:

```
# Less readable
def calculate_total(price,quantity): return price*quantity

# More readable
def calculate_total(price, quantity):
    return price * quantity
```

PEP 8 is a style guide, not a Python compiler rule.

Python will often run code that does not follow PEP 8.

Four Spaces for Indentation

Python convention uses four spaces for each indentation level.

Recommended:

```
if user_is_active:
    print("Active")
```

Avoid mixing tabs and spaces.

Consistent indentation makes nested code much easier to understand.

Spaces Around Operators

Readable:

```
total = price * quantity
```

Less readable:

```
total=price*quantity
```

For example:

```
age = 25
is_adult = age >= 18
```

Spacing helps visually separate the parts of an expression.

Spaces After Commas

Prefer:

```
numbers = [1, 2, 3, 4]
```

instead of:

```
numbers = [1,2,3,4]
```

Likewise:

```
def create_user(name, age, active):
    ...
```

Blank Lines

Blank lines help separate logical sections.

For example:

```
class User:
    ...


class Resource:
    ...
```

```
def create_resource():
    ...
```

They make the structure easier to scan.

Line Length

PEP 8 traditionally recommends keeping lines around 79 characters.

Modern projects may use a different limit, often 88 or another configured value.

The important principle is:

> Use a consistent line-length rule across the project.

Instead of:

```
result = calculate_total(price, quantity, discount, tax, shipping, currency)
```

you can write:

```
result = calculate_total(
    price,
    quantity,
    discount,
    tax,
    shipping,
    currency,
)
```

Naming Conventions

Python style conventions commonly use:

Variables

```
user_name = "Hafsa"
```

Functions

```
def calculate_total():
    ...
```

Classes

```
class ResourceManager:
    ...
```

Constants

```
MAX_RETRIES = 3
```

Private/internal names

```
_internal_value = 10
```

Consistent naming makes code easier to navigate.

Imports

Imports should generally be organized clearly.

Example:

```
import json
import os

from pathlib import Path

from myapp.models import Resource
```

A common structure is:

```
Standard library
        ↓
Third-party packages
        ↓
Local application imports
```

Import sorting tools can automate this.

What Is a Linter?

A linter is a tool that analyzes source code and reports potential issues.

A linter can detect things such as:

- unused imports
- unused variables
- undefined names
- suspicious expressions
- unreachable code
- unnecessary complexity

- inconsistent patterns
- style violations
- some common bugs

For example:

```
import json

def calculate_total(price, quantity):
    unused_value = 100
    return price * quantity
```

A linter may identify:

```
unused import: json
unused variable: unused_value
```

Popular Python Linters

Several tools are commonly used in Python projects.

Important ones include:

- Ruff
- Pylint
- Flake8

The Python ecosystem changes over time, so projects should choose tools based on their current needs and supported configuration.

For a modern project, **Ruff** is particularly useful because it combines fast linting with formatting capabilities.

Ruff

Ruff is a fast Python linter and formatter.

It can handle many tasks that historically required several separate tools.

Install it with:

```
pip install ruff
```

Check the version:

```
ruff --version
```

Running Ruff

To lint a project:

```
ruff check .
```

To automatically fix supported lint issues:

```
ruff check . --fix
```

To format code:

```
ruff format .
```

So a basic workflow can be:

```
ruff check .
      ↓
Find lint issues

ruff check . --fix
      ↓
Fix supported issues

ruff format .
      ↓
Format code
```

Ruff Formatter

Ruff can format Python files automatically.

Before:

```
def add(a,b):
 return a+b
```

After formatting:

```
def add(a, b):
    return a + b
```

Run:

```
ruff format .
```

The formatter applies a consistent style across the project.

Formatter vs Linter in Ruff

These commands do different things.

Lint

```
ruff check .
```

Looks for lint issues.

Auto-fix lint issues

```
ruff check . --fix
```

Attempts to fix supported issues.

Format

```
ruff format .
```

Reformats code consistently.

Think:

```
check → identify quality issues
fix   → automatically fix supported issues
format → normalize code appearance
```

Why Use an Automatic Formatter?

Imagine a team of five developers.

Without an automatic formatter, developers may disagree about:

```
quotes
spacing
line breaks
indentation
trailing commas
line length
```

With a formatter, the project has one consistent style.

Instead of arguing about formatting during code review:

```
Developer
   ↓
Write code
   ↓
```

```
Formatter
   ↓
Consistent output
```

Formatting becomes automated.

Formatters Reduce Review Noise

Suppose a pull request contains:

```
200 lines of actual changes
+
500 lines of formatting changes
```

Review becomes harder.

A shared formatter reduces unnecessary formatting differences.

Then reviewers can focus more on:

- logic
- architecture
- behavior
- security
- correctness

Should Formatting Be Manual?

For modern projects, automatic formatting is generally preferable.

Instead of manually adjusting:

```
x=10
```

to:

```
x = 10
```

let the formatter handle it.

This reduces repetitive work.

What Does Linting Catch?

Consider:

```
def calculate_total(price, quantity):
    result = price * quantity
```

```
    return result
    print("Done")
```

The final `print()` is unreachable.

A linter may identify the problem.

Another example:

```
def greet(name):
    message = f"Hello {name}"
    return "Hello"
```

The variable `message` is unused.

A linter can flag it.

Undefined Names

Consider:

```
def calculate():
    return price * quantity
```

If `price` and `quantity` are not defined in the available scope, a linter can detect the undefined names.

This is useful because such mistakes may otherwise become runtime errors.

Unused Imports

Example:

```
import json
import os

print(os.getcwd())
```

If `json` is never used, a linter can report it.

Unused imports create unnecessary clutter and can sometimes hide more important issues.

Unused Variables

Example:

```
def calculate():
    total = 100
```

```
    message = "Done"

    return total
```

`message` is never used.

A linter can flag it.

Duplicate or Suspicious Code

Depending on the tool and enabled rules, linters can identify suspicious patterns such as:

- duplicated expressions
- unnecessary conditions
- redundant code
- questionable comparisons
- overly complex expressions

Linting does not prove that code is correct.

It highlights patterns worth reviewing.

Lint Rules

A linter contains many individual rules.

Each rule targets a particular pattern.

For example:

```
unused import
undefined name
unused variable
line too long
bad naming
unnecessary expression
```

Tools assign identifiers to many rules.

Ruff, for example, uses rule codes such as:

```
F401
F841
E501
```

The exact enabled rules depend on configuration.

Why Rule Codes Matter

Suppose you see:

```
F401 imported but unused
```

The code:

```
F401
```

identifies the specific lint rule.

This allows you to:

- understand the warning
- search documentation
- selectively ignore a rule
- configure the project

Enabling Rules

A project can configure which lint rules should apply.

For example, Ruff can be configured in:

```
pyproject.toml
```

Example:

```
[tool.ruff.lint]
select = ["E", "F"]
```

This tells Ruff which rule families to enable.

Configuration should reflect the needs of the project rather than blindly enabling every possible rule.

Ignoring Specific Rules

Sometimes a project intentionally allows a pattern.

A rule can be ignored through configuration.

For example:

```
[tool.ruff.lint]
ignore = ["E501"]
```

This tells the linter not to report that particular rule.

Use ignores intentionally.

Do not use them simply to make the linter output disappear.

Inline Ignore

A specific line can sometimes include an inline suppression:

```
value = legacy_function()  # noqa: F401
```

The exact syntax depends on the tool.

Inline suppressions should be rare and justified.

Good Linting Philosophy

A useful principle is:

> A warning should either be fixed or intentionally explained.

Bad workflow:

```
Warning
   ↓
Ignore everything
```

Better workflow:

```
Warning
   ↓
Understand it
   ↓
Fix it
   OR
Intentionally configure/ignore it
```

Ruff Configuration With `pyproject.toml`

A project can centralize formatting and linting configuration.

Example:

```
[tool.ruff]
line-length = 88
target-version = "py312"

[tool.ruff.lint]
```

```
select = ["E", "F"]

[tool.ruff.format]
quote-style = "double"
```

This gives the project a shared configuration.

The exact settings should match the Python version and project conventions.

Why `pyproject.toml` Is Useful

Instead of remembering commands and rules manually, configuration lives with the project.

For example:

```
project/
├── pyproject.toml
├── src/
├── tests/
└── README.md
```

The project now documents its tooling expectations.

A new developer can clone the repository and use the same configuration.

Formatting and Linting With Virtual Environments

Since formatting and linting tools are development dependencies, they can be installed inside the project's virtual environment.

For example:

```
python -m venv .venv
```

Activate it and install:

```
pip install ruff
```

This keeps project tooling isolated.

Do not rely on random globally installed versions when a project needs reproducible tooling.

Development Dependencies

A project may have dependencies that are needed only during development.

Examples:

```
ruff
pytest
mypy
```

These are different from application dependencies such as:

```
requests
fastapi
django
```

A project can organize these using its chosen dependency-management approach.

Formatting Before Committing

A useful workflow is:

```
ruff format .
ruff check .
```

Then inspect the changes and commit.

The order can vary, but the important point is to format and lint before code reaches review.

Pre-Commit Hooks

Formatting and linting can be automated with Git hooks.

A pre-commit hook can run checks before a commit is created.

Conceptually:

```
git commit
    ↓
pre-commit
    ↓
formatter
    ↓
linter
    ↓
commit allowed
```

This reduces the chance of committing obvious quality problems.

CI Checks

Local checks are useful, but CI provides another layer.

Example:

```
Developer
   ↓
git push
   ↓
CI
   ├── Ruff format check
   ├── Ruff lint
   ├── mypy
   └── pytest
   ↓
Pass / Fail
```

This prevents the repository from relying entirely on each developer remembering to run the tools.

Format Check in CI

Instead of changing files in CI, you can ask Ruff whether files are already formatted.

For example:

```
ruff format --check .
```

This is useful in automated pipelines.

Local development can use:

```
ruff format .
```

to actually format files.

Lint Check in CI

Use:

```
ruff check .
```

If Ruff reports a configured error, the CI job can fail.

This creates a quality gate.

Formatting, Linting, Type Checking, Testing

These four tools have different jobs.

Formatter

```
How should code look?
```

Linter

```
What code patterns look suspicious or problematic?
```

Type checker

```
Are values being used according to their types?
```

Tests

```
Does the application behave correctly?
```

A strong workflow combines them:

```
Code
 ↓
Format
 ↓
Lint
 ↓
Type Check
 ↓
Test
 ↓
Commit
 ↓
CI
```

Ruff vs Black

Black is a dedicated Python formatter.

Ruff also provides a formatter.

Historically, a project might use:

```
Black → formatting
Flake8 → linting
isort → import sorting
```

A modern setup can consolidate many of these responsibilities around Ruff.

The right choice depends on an existing project's tooling and team preferences.

Ruff vs Flake8

Flake8 is a long-established Python linting tool.

Ruff is designed to be much faster and covers many linting rules that previously required multiple plugins.

Existing projects may already depend on Flake8, while newer projects may choose Ruff.

The important thing is consistency rather than switching tools without a reason.

Ruff vs Pylint

Pylint provides extensive static analysis and configurable checks.

Ruff focuses heavily on speed and compatibility with many common linting rules.

Neither tool makes a project automatically high quality.

The quality comes from:

```
appropriate rules
+
consistent configuration
+
developer judgment
```

Does Formatting Change Program Behavior?

A correct formatter should change code layout rather than the intended behavior.

For example:

```
total=price*quantity
```

becomes:

```
total = price * quantity
```

The intended computation remains the same.

However, formatting tools should still be used on committed code carefully and reviewed when making large changes.

Formatting Is Not Refactoring

Formatting:

```
x=10
```

to:

```
x = 10
```

is formatting.

Changing:

```
def calculate_total(...):
```

into a completely different architecture is refactoring.

A formatter should not be expected to redesign your code.

Linting Is Not Testing

A linter may identify:

```
unused_variable = 10
```

But a linter generally cannot determine whether:

```
def calculate_discount(price):
    return price * 0.5
```

contains the correct business logic for your application.

Tests are needed to verify behavior.

Linting Is Not Type Checking

Consider:

```
def add(a: int, b: int) -> int:
    return a + b
```

Calling:

```
add("10", 20)
```

is primarily a type-checking concern.

mypy can detect the incompatible argument.

A linter may not be responsible for detecting it.

Use the appropriate tool for each problem.

Example: Complete Development Toolchain

A Python project could use:

```
Ruff
  ↓
Formatting + linting

mypy
  ↓
Static type checking

pytest
  ↓
Automated tests
```

Example commands:

```
ruff format .
ruff check .
mypy .
pytest
```

This creates a simple quality pipeline.

Real World Example: haas.dev Resource Service

Imagine a Python service responsible for managing haas.dev resources.

Before formatting:

```
def find_resources(query,limit=10):
 results=[]
 for resource in resources:
  if query.lower() in resource.title.lower():
   results.append(resource)
 return results[:limit]
```

Formatted:

```
def find_resources(query, limit=10):
    results = []

    for resource in resources:
        if query.lower() in resource.title.lower():
            results.append(resource)
```

```
    return results[:limit]
```

Then add type hints:

```
def find_resources(
    query: str,
    limit: int = 10,
) -> list[Resource]:
    results: list[Resource] = []

    for resource in resources:
        if query.lower() in resource.title.lower():
            results.append(resource)

    return results[:limit]
```

Now different tools can contribute:

```
Ruff
 ↓
format + lint

mypy
 ↓
check types

pytest
 ↓
check behavior
```

A Practical Project Setup

A simple modern Python project could look like:

```
resource_app/
│
├── src/
│   └── resource_app/
│       ├── __init__.py
│       ├── models.py
│       ├── services.py
│       └── repositories.py
│
```

```
├── tests/
│   ├── test_models.py
│   └── test_services.py
│
├── pyproject.toml
├── README.md
└── .gitignore
```

The `pyproject.toml` can hold tool configuration.

Example `pyproject.toml`

```
[tool.ruff]
line-length = 88
target-version = "py312"

[tool.ruff.lint]
select = ["E", "F"]

[tool.mypy]
python_version = "3.12"
check_untyped_defs = true
disallow_untyped_defs = true
warn_return_any = true
```

The exact configuration can evolve as the project grows.

Team Workflow

A team can establish a simple rule:

```
Before opening a pull request:

1. Format
2. Lint
3. Type check
4. Run tests
```

For example:

```
ruff format .
ruff check .
```

```
mypy .
pytest
```

Then create the pull request.

CI runs the same checks independently.

Common Mistakes

Mistake 1: Formatting Manually

Do not spend time manually fixing every whitespace issue.

Use a formatter.

Mistake 2: Treating Every Lint Rule as Mandatory

Rules are tools, not absolute laws.

Choose rules appropriate for your project.

Mistake 3: Ignoring Every Warning

A warning can indicate a real problem.

Understand it before suppressing it.

Mistake 4: Mixing Multiple Formatters

Avoid using several formatters that disagree about formatting.

Choose one primary formatter.

Mistake 5: Different Developer Configurations

If every developer uses different settings, formatting changes can constantly appear in pull requests.

Keep configuration in the repository.

Mistake 6: Running Checks Only Before Release

Quality tools are most useful throughout development.

Mistake 7: Formatting the Entire Repository Accidentally

Before running automatic formatting on an existing project, understand which files are included.

A huge formatting-only diff can make review difficult.

Mistake 8: Suppressing Problems Instead of Fixing Them

Avoid turning:

```
warning
```

into:

```
ignore everything
```

unless there is a deliberate reason.

Best Practices

1. Choose one formatter

For example:

```
Ruff formatter
```

or:

```
Black
```

Do not let multiple formatters fight each other.

2. Keep configuration in the repository

Use:

```
pyproject.toml
```

where appropriate.

3. Automate checks

Use:

```
pre-commit
CI
```

4. Keep lint rules intentional

Enable rules that provide meaningful value.

5. Fix warnings early

Small warnings become harder to manage when accumulated across a large project.

6. Combine tools

A formatter alone is not enough.

Consider:

```
Formatter
+
Linter
+
Type checker
+
Tests
```

7. Keep CI consistent with local development

Developers should be able to reproduce CI checks locally.

---

5 Minute Challenge

Create a Python file containing intentionally messy code:

```
import os
import json

def calculate(a,b):
 x=a+b
 unused="hello"
 return x

print(calculate(10,20))
```

Run:

```
ruff format .
```

Then:

```
ruff check .
```

Identify:

- formatting problems
- unused imports
- unused variables
- any other lint warnings

Fix the remaining issues.

Then run:

```
ruff check .
```

again until the code passes the configured rules.

Exercises

Exercise 1: Format a Function

Start with:

```
def calculate_total(price,quantity,discount=0):
 total=price*quantity
 return total-(total*discount)
```

Format it manually first.

Then run Ruff and compare your formatting with the formatter's result.

Exercise 2: Find Lint Problems

Create:

```
import os
import json

def greet(name):
    unused_value = 10
    return "Hello " + name
```

Run:

```
ruff check .
```

Identify each warning.

Exercise 3: Configure Ruff

Create a `pyproject.toml` and configure:

```
line length
target Python version
selected lint rules
```

Run Ruff again.

Exercise 4: Build a Quality Command List

Create a simple development routine:

```
ruff format .
ruff check .
mypy .
pytest
```

Run each command against a small project.

Record what each tool reports.

Exercise 5: CI Simulation

Before pushing code:

```
ruff format --check .
ruff check .
mypy .
pytest
```

Treat any failure as a reason to stop and fix the project before committing.

Mini Project: Clean Python Resource Manager

Build a small Python resource manager.

Create:

```
class Resource:
    ...
```

with:

```
title
category
pages
```

Then create functions:

```
def add_resource(...):
    ...

def find_resource(...):
    ...

def list_resources(...):
    ...
```

Requirements:

Formatting

Use Ruff to format all files.

Linting

Use Ruff to identify:

- unused imports
- unused variables
- undefined names
- suspicious patterns

Type Checking

Use mypy for:

- function parameters
- return types
- resource objects
- optional results

Testing

Use pytest to test:

- adding resources
- searching resources
- missing resources
- empty collections

Your final workflow should be:

```
Write code
   ↓
ruff format
   ↓
ruff check
   ↓
mypy
   ↓
pytest
   ↓
Commit
```

Interview Questions

1. What is code formatting?

Code formatting is the automatic or manual organization of source code according to consistent style rules.

2. What is linting?

Linting analyzes source code for potential errors, suspicious patterns, style problems, and maintainability issues.

3. What is PEP 8?

PEP 8 is Python's widely used style guide describing conventions for readable Python code.

4. What is Ruff?

Ruff is a fast Python linter that also provides a formatter.

5. What does this command do?

```
ruff format .
```

It formats Python files in the specified project path.

6. What does this command do?

```
ruff check .
```

It checks the project for configured linting issues.

7. What does `--fix` do?

```
ruff check . --fix
```

It asks Ruff to automatically fix supported lint violations.

8. Is formatting the same as linting?

No.

Formatting focuses primarily on code appearance; linting focuses on potential problems and code quality.

9. Is linting the same as testing?

No.

Linting analyzes source code patterns; testing executes code to verify behavior.

10. Is linting the same as type checking?

No.

Type checking focuses on type relationships, while linting covers a broader range of code-quality patterns.

11. Why use automatic formatting?

It provides consistent code style and reduces unnecessary formatting discussions during code review.

12. Why configure tools in `pyproject.toml`?

It provides a central, version-controlled location for project tooling configuration.

13. Why use CI for linting?

CI ensures that code meets the project's quality checks before it is merged or deployed.

14. Should every lint warning be ignored?

No. Warnings should normally be fixed or intentionally suppressed with a clear reason.

15. Why should a project avoid multiple competing formatters?

Different formatters may produce conflicting changes, creating unnecessary formatting churn.

Code Quality Cheat Sheet

Format

```
ruff format .
```

Check formatting

```
ruff format --check .
```

Lint

```
ruff check .
```

Auto-fix supported lint issues

```
ruff check . --fix
```

Install Ruff

```
pip install ruff
```

Install mypy

```
pip install mypy
```

Type check

```
mypy .
```

Run tests

```
pytest
```

Typical local workflow

```
ruff format .
ruff check .
mypy .
pytest
```

Example `pyproject.toml`

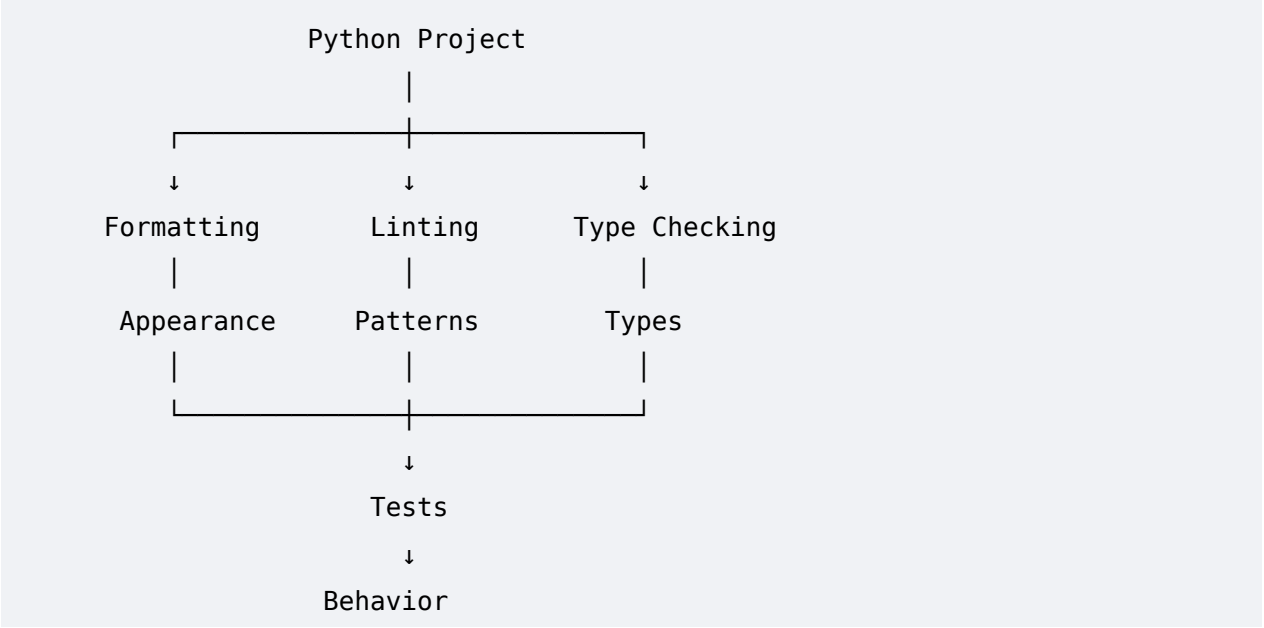
```
[tool.ruff]
line-length = 88
target-version = "py312"

[tool.ruff.lint]
select = ["E", "F"]

[tool.mypy]
python_version = "3.12"
```

Final Mental Model

Think of code quality as several separate layers:



Each layer answers a different question.

Formatter

> Does the code follow a consistent visual style?

Linter

> Does the code contain suspicious or undesirable patterns?

Type checker

> Are values being used consistently with their declared types?

Tests

> Does the program actually behave correctly?

A professional Python workflow does not depend on any single tool.

The goal is to automate the repetitive checks so developers can spend more time solving actual problems.

Visit haas.dev for more resources and guides.

https://dev-roast-app.vercel.app

https://dev-roast-app.vercel.app/resources