Beyond PEP-8: A Formatting Style for Visual Parsing

A guide to a Python style that prioritises information density and pattern recognition.

JetBrains Mono, Regular
Version: v3.63.4 | **Updated:** January 2026 | **Related:** Type_Safe Capabilities Guide

Code is read more often than it is written.

Standard Python formatting is optimised for writability. This guide is optimised for readability, debugging, and review. We trade minor formatting effort for major long-term gains in clarity and speed by treating code as a visual interface.

Visual Pattern Recognition

Humans are brilliant at spotting patterns. Consistent vertical alignment makes code structure, relationships, and even bugs, immediately apparent.

Reduced Cognitive Load

By grouping related information and creating clear ‘visual lanes’, we reduce the mental effort required to parse code, freeing up brainpower for logic.

Information Density

Less vertical space and fewer distracting artifacts (like docstrings) mean more meaningful code is visible on screen at once, preserving context.

Establishing a Clear Blueprint

The File Header

Every file must begin with a header comment, using exactly 80 `=` characters for the border, before any imports.

```
# ================================================================
# Title of the File
# Optional subtitle or description of the file's purpose
# ================================================================

import ...
```

Section Dividers

Use formatted comment dividers to organise logical groups of methods or classes within a file.

```
    # ... some methods ...

    # {Title} - {Subtitle}

    class AnotherClass:
        ...
```

Creating Visual Lanes with Aligned Imports

- One import per line.
- Align the `import` keyword to a consistent column (e.g., 80-85).
- Group by: 1. Standard library, 2. Third-party, 3. Local modules. Separate groups with a blank line.

Before: Standard PEP-8

```
# BEFORE
import os
import sys
from collections import defaultdict

import pandas as pd
import requests

from my_project.utils import helper_function
from my_project.models.user import User
```

After: Aligned for Clarity

```
# AFTER
import os
import sys
from collections             import defaultdict

import pandas                as pd
import requests

from my_project.utils        import helper_function
from my_project.models.user  import User
```

The End of Docstrings

CRITICAL: NEVER use Python docstrings in Type_Safe code.

The Rationale

- Docstrings add significant vertical noise and separate documentation from the code it describes.
- In a type-annotated codebase, the signature itself provides the ‘what’. The ‘why’ and ‘how’ belong in inline comments, directly adjacent to the implementation.

The Alternative

All documentation must be inline comments, aligned at a consistent column (e.g., 70-80).

```
def process_data(self, user: User) -> ProcessedData:      # Processes raw user data into a canonical format.
```

The Anatomy of an Aligned Method Signature

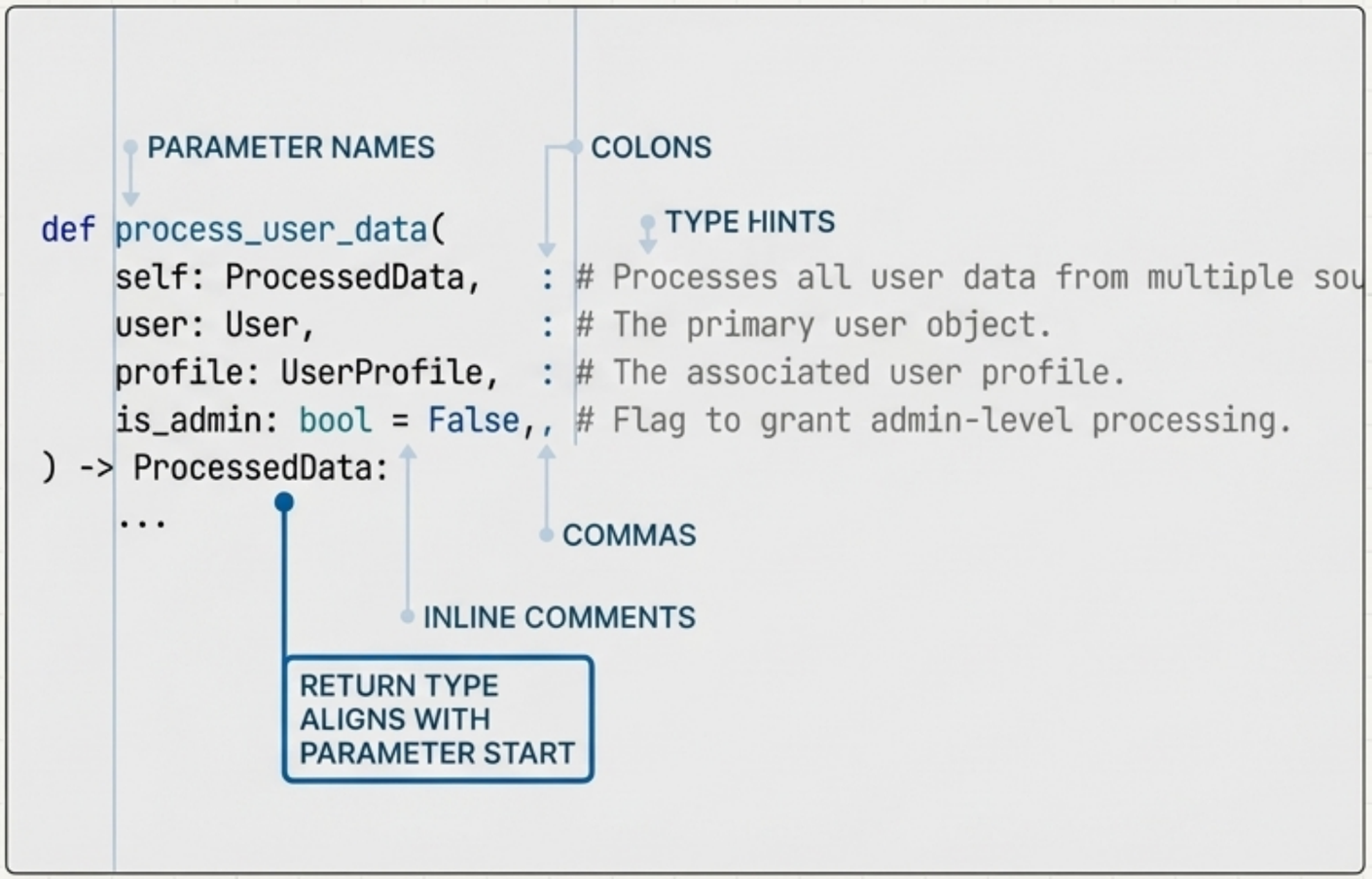
Stack parameters vertically to treat the signature like a structured table of information.

Critical Alignment Rules

1. The first parameter remains on the same line as the method name. Subsequent parameters are stacked vertically, aligned with the first.

2. Align parameter names, colons (`:`), type annotations, commas (`,`), and inline comments (`#`) into distinct columns.
3. **Key Alignment**: The first letter of the return type must align perfectly with the first letter of the parameter names.

4. The `self` parameter's comment describes the method's overall purpose.



Pragmatism in Practice: Simple Signatures and Calls

Single-Parameter Methods

Complex vertical alignment is unnecessary for simple signatures. Keep them on a single line.

```python
# DO
def get_user_by_id(self, user_id: int) -> User:
    ...
```

The First Parameter Rule

CRITICAL: Never put a line break before the first parameter. This applies to both method definitions and calls. It ensures the name of the function/method and its first argument are always visually connected.

AVOID vs. Do

```python
# AVOID
user = create_user(
    name="John Doe",
    email="john.doe@example.com",
)
```

DO

```python
# DO
user = create_user(name="John Doe",
                   email="john.doe@example.com")
```

Editorial Blueprint

Aligning Operators to Reveal Structure

In blocks of related assignments or assertions, aligning the operators (`=`, `==`, `is`, etc.) turns the code into an easily scannable table, making it simple to verify logic.

1. Aligned Assignments

```
# A block of related variable assignments
self.host      = "localhost"
self.port      = 5432
self.username  = os.getenv("DB_USER")
self.db_name   = "primary_data_store"
```

2. Aligned Assertions

```
# A block of test assertions
assert result.status_code == 200
assert result.user_id     == expected_user.id
assert result.is_active   is True
```

3. Aligned Boolean Checks

```
# A block of pre-condition checks
is_valid_user    = user is not None
has_permissions  = user.role == "admin"
is_enabled       = feature_flags.is_active("new_feature")
```

Editorial Blueprint

Bringing Order to Constructors

Rules for Multi-Parameter Constructors

1. The first parameter stays on the same line as the class name.
2. Subsequent parameters are aligned with the first.
3. Align any `=` signs within the call.
4. The closing parenthesis `)` goes on the same line as the last parameter.

```
new_user = User(
    name="Jane Doe",
    email="jane.doe@example.com",
    role=UserRole.EDITOR,
    send_welcome_email=True,
)
```

Visual 1: Standard Constructor Call

```
report = FinancialReport(
    report_id=generate_id(),
    period=DateRange(
        start_date=datetime(2026, 1, 1),
        end_date=datetime(2026, 1, 31)),
    author=self.current_user,
)
```

Visual 2: Nested Constructor Call

Editorial Blueprint

Class Definitions and Naming Philosophy

No Private Method Prefixes

CRITICAL: Do not use a leading underscore (`_`) for helper/private methods.

In Type_Safe classes, all methods are considered part of the class's public interface. The underscore convention adds visual noise without providing meaningful encapsulation in this context.

A Taxonomy of Class Types

Schema Classes

Pure data containers. No methods. Use for data transfer objects.

Collection Subclasses

Pure type definitions for collections. No methods.
(e.g., `class UserIdList(list[UserId]): ...`)

Service/Helper Classes

Contain the business logic. This is where all methods reside.

Consistency in Types and Control Flow

Class Attribute Annotations

Align colons and default values for class attributes.

```
class Config:
    timeout: int = 10
    retries: int = 3
    host:    str = "api.example.com"
```

Guard Clauses

Use early returns for validation to reduce nested `if` statements.

```
def process_item(item: Item | None) -> None:
    if item is None:
        return
    # ... proceed with processing
```

Explicit Boolean Comparisons

Always use `is True` or `is False` for clarity. Avoid implicit boolean checks (e.g., `if is_valid:`).

DO ✓

```
# DO
if user.is_active is True:
    ...
```

AVOID ✗

```
# AVOID
if user.is_active:
    ...
```

Formatting Data Structures

Section 1: Dictionaries

For multi-line dictionary literals and comprehensions, align keys and values to create a clear table-like structure.

```
user_data = {
    "user_id":   123,
    "username":  "j.doe",
    "is_active": True,
}
```

```
{
    user.id: user.name
    for user in active_users
        if user.signup_date > threshold
}
```

Section 2: Lists

Keep simple list comprehensions on one line. For multi-line list literals, align elements if it enhances clarity.

```
SUPPORTED_MODES = [
    "READ_ONLY",
    "READ_WRITE",
    "ADMIN",
]
```

Structure and Clarity in Test Code

Test Method Structure

Organise every test method into three distinct, commented blocks: Setup, Execute, and Verify. This creates a consistent and readable pattern across the entire test suite.

```
def test_user_deactivation_succeeds(self):
    # Setup
    user_service = self.get_user_service()
    active_user  = self.create_active_user(name="test")

    # Execute
    result = user_service.deactivate(user_id=active_user.id)

    # Verify
    assert result.is_success is True
    updated_user = user_service.get_by_id(active_user.id)
    assert updated_user.is_active is False
```

Anti-Patterns: A Gallery of What to Avoid

AVOID: Docstrings

```python
def my_function(a: int) -> int:
    """This is a docstring. Avoid it."""
    ...
```

AVOID: Underscore Prefixes

```python
class MyService:
    def _internal_helper(self): …
```

AVOID: Line Break Before First Parameter

```python
result = my_func(
    param1=value1,
)
```

AVOID: Methods in Schema Classes

```python
class UserData(Schema):
    name: str
    def get_name(self): ...
```

AVOID: Implicit Boolean Checks

```python
if user.is_active: ...
```

Your Formatting Checklist

Use this checklist during code authoring and review to ensure adherence to the style guide.

- [ ] File header with `=` borders is present.
- [ ] Logical sections are separated by dividers.
- [ ] All imports are aligned at a consistent column.
- [ ] **No docstrings exist.** All documentation is inline.
- [ ] Inline comments are aligned.
- [ ] Multi-parameter method signatures are vertically aligned.
- [ ] Return type's first letter aligns with parameter names.
- [ ] No line break before the first parameter in any call.
- [ ] No `_` prefix on helper methods.
- [ ] Related assignment `=` signs are aligned.
- [ ] Comparison operators in assertion blocks are aligned.
- [ ] Boolean checks use explicit `is True` / `is False`.
- [ ] Schemas and Collection subclasses contain no methods.
- [ ] Guard clauses are used for early returns.
- [ ] Test methods use the Setup, Execute, Verify structure.