Writing Maintainable Python

Build Python Code That Stays Easy to Understand, Test, and Change

Writing Python code that works is only the beginning.

A small script can be written quickly:

```
x = 100
y = 20
z = x - y
print(z)
```

But real projects grow.

A few functions become dozens.

A few files become modules.

One developer becomes a team.

A feature that originally took one hour may eventually require changes across five different parts of the application.

This is where **maintainability** matters.

Maintainable Python is code that remains reasonably easy to:

- read
- understand
- test
- debug
- modify
- extend
- review
- reuse
- remove

The goal is not to write the shortest possible code.

The goal is to write code that your future self and other developers can understand without fighting the codebase.

---

1. What Does Maintainable Code Mean?

Imagine two functions.

```
def f(a, b, c):
    return a * b - c
```

And:

```
def calculate_remaining_budget(income, expenses, savings):
    return income - expenses - savings
```

Both may work.

But the second version communicates intent.

A developer reading it does not have to figure out what `a`, `b`, and `c` represent.

Maintainability is largely about reducing the amount of mental effort required to understand and safely change code.

A useful mental model is:

```
Maintainable Code
        ↓
Easy to understand
        +
Easy to test
        +
Easy to change
        +
Hard to misuse
```

---

2. Working Code vs Maintainable Code

Code can be correct and still be difficult to maintain.

For example:

```
def process(data):
    x = []
    for i in data:
        if i["active"] == True:
            if i["price"] > 0:
                x.append(i["price"] * 0.9)
    return x
```

This may work.

But what does it actually do?

A clearer version:

```
DISCOUNT_RATE = 0.10


def calculate_discounted_prices(products):
    discounted_prices = []

    for product in products:
        if product["active"] and product["price"] > 0:
            discounted_price = product["price"] * (1 - DISCOUNT_RATE)
            discounted_prices.append(discounted_price)

    return discounted_prices
```

The second version exposes the intent.

---

3. The Three Questions of Maintainability

When writing Python, ask:

Can another developer understand this?

If not, improve naming, structure, or documentation.

Can I change one part without breaking unrelated parts?

If not, reduce coupling.

Can I test this behavior easily?

If not, consider simplifying responsibilities and dependencies.

These questions are more useful than simply asking:

> "Does this code work?"

---

4. Principle 1: Use Clear Names

Names are one of the cheapest ways to improve maintainability.

Avoid:

```
x = 10
y = 20
z = x + y
```

Prefer:

```
monthly_income = 10
monthly_expenses = 20
remaining_balance = monthly_income - monthly_expenses
```

The second version contains information directly in the code.

---

5. Names Should Explain Purpose

Poor:

```
def process(user):
    ...
```

Better:

```
def create_user_profile(user):
    ...
```

Poor:

```
data = get_data()
```

Better:

```
resources = get_resources()
```

Poor:

```
result = calculate(items)
```

Better:

```
total_price = calculate_total_price(items)
```

A good name reduces the need for comments.

---

6. Avoid Meaningless Names

Names such as:

```
data
thing
stuff
temp
foo
bar
value
result
```

are not always wrong.

They become problematic when they hide important meaning.

For example:

```
data = get_user_data()
```

might be acceptable for a tiny local scope.

But:

```
data = get_data()
```

in a 500-line service module tells the reader almost nothing.

Prefer:

```
users = get_users()
```

or:

```
resource_records = get_resource_records()
```

---

7. Use Names at the Right Level

A variable should communicate what matters in its context.

Instead of:

```
for x in resources:
    print(x.title)
```

use:

```
for resource in resources:
    print(resource.title)
```

But do not overdo it:

```
for resource_object_instance_from_database in resources:
    ...
```

Good names are descriptive without becoming unnecessarily long.

---

8. Constants Should Have Names

Avoid magic numbers:

```
price = price * 0.85
```

Prefer:

```
DISCOUNT_RATE = 0.15

price = price * (1 - DISCOUNT_RATE)
```

Now the meaning is visible.

Other examples:

```
MAX_LOGIN_ATTEMPTS = 3
DEFAULT_PAGE_SIZE = 20
REQUEST_TIMEOUT_SECONDS = 10
```

---

9. Avoid Magic Strings

Instead of:

```
if user.role == "admin":
    ...
```

you may use:

```
ADMIN_ROLE = "admin"

if user.role == ADMIN_ROLE:
    ...
```

For larger systems, an enum may be more appropriate:

```
from enum import Enum


class UserRole(Enum):
    ADMIN = "admin"
    USER = "user"
```

Then:

```
if user.role == UserRole.ADMIN:
    ...
```

The right choice depends on project complexity.

---

10. Keep Functions Focused

A function should ideally have one clear responsibility.

Consider:

```
def create_resource(data):
    validate(data)
    save_to_database(data)
    send_email(data)
    write_log(data)
    update_cache(data)
```

This function has many responsibilities.

A better design might separate them:

```
def create_resource(data):
    validate_resource(data)
    resource = save_resource(data)
    notify_resource_created(resource)
    return resource
```

The lower-level responsibilities can live elsewhere.

The goal is not "one line per function."

The goal is **clear responsibility**.

---

11. Function Length Is a Signal, Not a Rule

There is no universal rule saying:

> Every function must contain exactly 10 lines.

A 30-line function can be maintainable.

A 5-line function can be confusing.

For example:

```
def calculate_total(items):
    return sum(
        item.price
        for item in items
        if item.active
        and item.category != "archived"
        and item.stock > 0
    )
```

This is short but may become difficult to understand as logic grows.

Sometimes explicit steps are clearer:

```
def calculate_total(items):
    available_items = [
        item
        for item in items
        if item.active
        and item.category != "archived"
        and item.stock > 0
    ]

    return sum(item.price for item in available_items)
```

Optimize for clarity, not line count.

---

12. Avoid Deep Nesting

Nested conditions can quickly become difficult to follow.

Harder to read:

```
def process_user(user):
    if user:
        if user.is_active:
            if user.email:
                if user.has_permission:
                    process(user)
```

Use early returns:

```
def process_user(user):
    if not user:
        return

    if not user.is_active:
        return

    if not user.email:
        return

    if not user.has_permission:
        return

    process(user)
```

The main behavior becomes easier to identify.

---

13. Guard Clauses

A guard clause exits early when a condition cannot continue.

Example:

```
def get_resource(resource_id):
    if not resource_id:
        return None

    resource = repository.find(resource_id)

    if resource is None:
        return None

    return resource
```

Instead of deeply nesting the main logic:

```
def get_resource(resource_id):
    if resource_id:
        resource = repository.find(resource_id)

        if resource:
            return resource

    return None
```

Both can work.

Guard clauses often make complex functions easier to scan.

---

14. Avoid Clever Code

Python allows concise expressions.

That does not mean every expression should be compressed.

Instead of:

```
active = [x for x in users if x.active and x.email]
```

this may be perfectly readable.

But if the condition becomes complicated:

```
valid = [
    x for x in users
```

```
    if x.active
    and x.email
    and x.role in allowed_roles
    and not x.is_suspended
    and x.last_login
]
```

a named helper can communicate intent:

```
def is_eligible_user(user, allowed_roles):
    return (
        user.active
        and user.email
        and user.role in allowed_roles
        and not user.is_suspended
        and user.last_login
    )


eligible_users = [
    user
    for user in users
    if is_eligible_user(user, allowed_roles)
]
```

Readable code beats clever code.

---

15. DRY: Don't Repeat Yourself

Suppose you have:

```
if user.age >= 18:
    ...
```

in six places.

This can become dangerous.

If the rule changes to 21, every location must be updated.

Create a reusable concept:

```
MINIMUM_ADULT_AGE = 18
```

```
def is_adult(user):
    return user.age >= MINIMUM_ADULT_AGE
```

Then:

```
if is_adult(user):
    ...
```

Now the rule has one obvious home.

---

16. But Don't Overuse DRY

DRY does not mean:

> Every repeated line must become a function.

For example, extracting a tiny expression too early can make code harder to understand.

Bad abstraction:

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

when the operation is already obvious.

The goal is to remove **duplicated knowledge**, not every repeated character.

A useful distinction:

```
Repeated code
      ≠
Repeated business knowledge
```

Business rules are especially valuable to centralize.

---

17. Prefer Composition Over Giant Functions

Instead of:

```
def process_order(order):
    # 100 lines
    # validation
    # pricing
    # payment
    # inventory
    # email
    # logging
```

separate responsibilities:

```
def process_order(order):
    validate_order(order)
    total = calculate_order_total(order)
    charge_payment(order, total)
    update_inventory(order)
    send_confirmation(order)
```

Now the high-level workflow is visible.

The details can be maintained independently.

---

18. Separate Business Logic From I/O

Consider:

```
def calculate_discount():
    user_input = input("Enter price: ")
    price = float(user_input)

    return price * 0.9
```

This function mixes:

- user interaction
- input parsing
- business logic

A more testable design:

```
def calculate_discounted_price(price):
    return price * 0.9


price = float(input("Enter price: "))
result = calculate_discounted_price(price)
print(result)
```

Now the calculation can be tested without simulating terminal input.

---

19. Separate Business Logic From Databases

Avoid putting all logic inside database code.

Harder:

```
def get_discounted_resources():
    rows = database.execute(
        "SELECT price FROM resources"
    )

    return [
        row["price"] * 0.9
        for row in rows
    ]
```

A better architecture can separate retrieval from calculation:

```
def calculate_discounted_prices(prices):
    return [price * 0.9 for price in prices]
```

Then another layer handles database access.

This reduces coupling.

---

20. Dependency Injection

Suppose a service creates its own database connection:

```
class ResourceService:
    def __init__(self):
        self.database = Database()
```

Testing becomes harder because the service controls its dependency.

Instead:

```
class ResourceService:
    def __init__(self, database):
        self.database = database
```

Now:

```
database = Database()
service = ResourceService(database)
```

A test can provide a fake dependency.

This is a practical maintainability technique.

---

21. Keep Modules Focused

A module containing everything becomes difficult to navigate.

Avoid:

```
app.py
```

with:

```
2000 lines
```

containing:

- database code
- authentication
- validation
- business logic
- API routes
- utilities
- configuration

Instead, organize related responsibilities:

```
project/
│
├── app.py
├── config.py
├── models.py
├── services.py
├── repositories.py
├── validators.py
└── utils.py
```

The exact structure depends on the project.

---

22. Avoid a Giant `utils.py`

A common mistake is creating:

```
utils.py
```

and eventually putting everything inside it.

You may end up with:

```
format_date()
calculate_price()
send_email()
```

```
parse_token()
resize_image()
validate_user()
generate_slug()
```

These functions have little relationship.

Prefer modules based on meaningful responsibilities:

```
dates.py
pricing.py
email.py
authentication.py
images.py
validation.py
```

The module name should help explain what belongs there.

---

23. Keep Imports Clean

Avoid unnecessary imports:

```
import os
import sys
import json
import math
import random
```

if only `json` is used.

Prefer:

```
import json
```

Clean imports make dependencies easier to understand.

Tools such as Ruff can detect many import and code-quality issues automatically.

---

24. Avoid Circular Dependencies

Imagine:

```
users.py
   ↓
orders.py
   ↓
users.py
```

This can produce circular import problems.

If two modules depend heavily on each other, it may indicate that responsibilities need to be redesigned.

Possible solutions include:

- extracting shared concepts
- moving a function
- introducing a service layer
- using dependency injection
- changing module boundaries

The correct solution depends on the architecture.

---

25. Use Type Hints for Clarity

Type hints communicate expected data.

Instead of:

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

use:

```
def calculate_total(items: list[float]) -> float:
    return sum(items)
```

This helps:

- readers
- IDEs
- static type checkers
- maintainers

Type hints do not automatically guarantee runtime correctness.

---

26. Use Docstrings When They Add Information

A docstring is useful when behavior is not obvious.

```
def calculate_shipping_cost(weight: float) -> float:
    """Calculate shipping cost based on package weight."""
    ...
```

Do not write pointless documentation:

```
def add(a, b):
    """Add a and b."""
    return a + b
```

if the function is already completely obvious.

Documentation should explain information that is useful to the reader.

---

27. Comments Should Explain Why

Bad comment:

```
# Add one to count
count += 1
```

The code already says that.

Better:

```
# The API uses 1-based page numbering.
page_number += 1
```

Comments should explain:

- why something is necessary
- unusual constraints
- external behavior
- non-obvious decisions

---

28. Don't Leave Dead Code

Dead code creates confusion.

For example:

```
def calculate_price(price):
    old_price = price * 0.8

    new_price = price * 0.9

    return new_price
```

`old_price` is unused.

Remove it.

Also remove:

- commented-out old implementations

- unused functions
- obsolete imports
- unreachable branches

Version control already preserves old code.

You do not need to keep old implementations commented out.

---

29. Avoid Premature Abstraction

Suppose you have:

```
def calculate_total(price):
    return price * 1.1
```

Do not immediately create:

```
pricing/
├── interfaces/
├── strategies/
├── factories/
├── managers/
└── adapters/
```

for a single calculation.

Start simple.

Add abstractions when real complexity appears.

A useful principle:

```
Simple first
     ↓
Understand the problem
     ↓
Identify repeated patterns
     ↓
Refactor when necessary
```

---

30. Refactoring

Refactoring means changing the internal structure of code without intentionally changing its external behavior.

Before:

```
def process(resources):
    active = []

    for resource in resources:
        if resource["published"]:
            active.append(resource)

    return active
```

After:

```
def is_published(resource):
    return resource["published"]


def get_published_resources(resources):
    return [
        resource
        for resource in resources
        if is_published(resource)
    ]
```

The external behavior can remain the same while the structure becomes easier to evolve.

---

31. Refactor With Tests

Refactoring without tests can be risky.

A safer workflow:

```
Existing behavior
       ↓
Write tests
       ↓
Refactor
       ↓
Run tests
       ↓
Compare behavior
```

If tests remain green, you have stronger evidence that behavior was preserved.

---

32. Maintainability and Testing

Testable code tends to have clearer boundaries.

For example:

```
def calculate_total(prices):
    return sum(prices)
```

is easy to test.

```
def calculate_total():
    database = Database()
    rows = database.query(...)
    send_email(...)
    print(...)
```

is harder to test because many unrelated concerns are combined.

Testing therefore acts as a design signal.

If a function is extremely difficult to test, ask:

> Does this function have too many responsibilities?

---

33. Error Handling Should Be Intentional

Avoid:

```
try:
    do_something()
except:
    pass
```

This hides failures.

Prefer specific exceptions:

```
try:
    amount = float(user_input)
except ValueError:
    print("Please enter a valid number.")
```

If an exception cannot be meaningfully handled at the current layer, allow it to propagate or handle it at a more appropriate boundary.

---

34. Don't Hide Important Failures

Bad:

```
try:
    save_resource(resource)
except Exception:
    return None
```

Now callers cannot easily distinguish:

- resource not found
- database failure
- validation failure
- programming bug

A better design defines meaningful error behavior.

For example:

```
try:
    save_resource(resource)
except DatabaseError as error:
    logger.error("Could not save resource: %s", error)
    raise
```

The exact handling depends on the application.

---

35. Logging Helps Maintainability

Maintainable applications need useful diagnostics.

Instead of:

```
print("something went wrong")
```

use logging:

```
import logging

logger = logging.getLogger(__name__)

logger.error("Failed to save resource")
```

Logging provides context without mixing debugging output with application behavior.

---

36. Configuration Should Not Be Scattered

Avoid:

```
API_URL = "https://example.com"
TIMEOUT = 10
MAX_RETRIES = 3
```

repeated throughout multiple modules.

Centralize configuration where appropriate:

```
class Settings:
    api_url = "https://example.com"
    timeout = 10
    max_retries = 3
```

For real applications, environment variables and configuration management can be used.

The important idea is to avoid mysterious duplicated configuration.

---

37. Use Small, Predictable Interfaces

A function should ideally have an understandable contract.

Example:

```
def calculate_total(prices: list[float]) -> float:
    ...
```

A caller can understand:

```
Input  → list of numbers
Output → number
```

Avoid functions whose behavior depends on hidden global state.

Harder:

```
current_user = None


def calculate_price():
    ...
```

Better:

```
def calculate_price(user, cart):
    ...
```

Explicit dependencies are easier to reason about.

---

38. Minimize Global State

Global mutable state can create surprising behavior.

For example:

```
users = []


def add_user(user):
    users.append(user)
```

Any part of the application can potentially modify `users`.

Prefer explicit ownership:

```
class UserRepository:
    def __init__(self):
        self.users = []

    def add(self, user):
        self.users.append(user)
```

Now the state has a clear owner.

---

39. Keep Data Flow Visible

Consider:

```
result = process(data)
```

If `process()` reads global variables, modifies a cache, updates a database, and changes another global object, the caller cannot easily understand what happened.

Explicit data flow is easier:

```
validated_data = validate(data)
processed_data = transform(validated_data)
result = save(processed_data)
```

The steps communicate the flow.

---

40. Maintainable Python Is Not Overengineered Python

There is an important difference.

Maintainable

```
def get_active_resources(resources):
    return [r for r in resources if r.active]
```

Overengineered

```
ResourceFilterFactory
ResourceFilterStrategy
AbstractResourceFilter
ResourceFilterProvider
```

for a simple filtering operation.

Good engineering uses the **simplest structure that handles the actual problem well**.

---

41. The Rule of Three

A useful refactoring heuristic:

```
First occurrence
→ write it simply

Second occurrence
→ notice the pattern

Third occurrence
→ consider extracting an abstraction
```

This prevents premature abstractions while still encouraging reuse.

It is a heuristic, not a strict law.

---

42. Maintainability and API Design

Suppose your application exposes:

```
def create_resource(title, description, category):
    ...
```

Changing it to:

```
def create_resource(title, description, category, author, status, tags):
    ...
```

may eventually become difficult to maintain.

A domain object or structured input may be more appropriate:

```
resource = ResourceInput(
    title=title,
    description=description,
    category=category,
    author=author,
    status=status,
    tags=tags,
)
```

The goal is to keep interfaces understandable as systems grow.

---

43. Backward Compatibility

Maintainable code considers existing users.

Suppose existing code calls:

```
get_resources(limit=20)
```

Changing the function unexpectedly can break many callers.

Before changing public interfaces, consider:

- who uses them
- whether compatibility matters
- whether a deprecation period is appropriate
- whether a new function is safer

This becomes especially important in libraries and APIs.

---

44. Code Review as a Maintainability Tool

Before merging code, ask:

Readability

Can I understand this quickly?

Responsibility

Does each function/module have a clear purpose?

Duplication

Is important logic repeated?

Coupling

Does this change force unrelated parts to change?

Testing

Can the important behavior be tested?

Error handling

Are failures handled intentionally?

Naming

Do names explain the domain?

Complexity

Is there unnecessary cleverness?

These questions make code review more useful than simply checking formatting.

---

45. A Maintainability Checklist

Before considering a feature complete:

```
[ ] Names are clear
[ ] Functions have focused responsibilities
[ ] Modules have meaningful boundaries
[ ] Important business rules are not duplicated
[ ] Global state is minimized
[ ] Dependencies are explicit
[ ] Errors are handled intentionally
[ ] Secrets are not hardcoded
[ ] Configuration is organized
[ ] Tests cover important behavior
[ ] Formatting is consistent
[ ] Linting passes
[ ] Dead code is removed
[ ] Comments explain why when necessary
[ ] Documentation exists where useful
[ ] Git history contains meaningful changes
```

---

46. A Real haas.dev Example

Imagine a Python service for haas.dev resources.

A poorly structured version might contain:

```
def handle_resource(data):
    # validate
    # save
    # search
    # format
```

```
    # notify
    # log
    # everything else
    ...
```

A more maintainable architecture could separate responsibilities:

```
haas_resources/
│
├── models.py
├── validators.py
├── repositories.py
├── services.py
├── search.py
├── notifications.py
└── logging_config.py
```

The service layer could coordinate the workflow:

```
def create_resource(data):
    resource = validate_resource(data)
    saved_resource = resource_repository.save(resource)
    notification_service.notify_created(saved_resource)

    return saved_resource
```

The exact architecture depends on project requirements.

The important idea is that each component has a clear responsibility.

---

47. Maintainability Workflow

Use this workflow when building a new feature:

```
Understand the requirement
        ↓
Choose a simple design
        ↓
Define clear data flow
        ↓
Write focused functions
        ↓
Use meaningful names
        ↓
```

```
Add tests
        ↓
Run formatter and linter
        ↓
Review the code
        ↓
Refactor obvious duplication
        ↓
Commit the change
```

Do not attempt to create the perfect architecture before understanding the problem.

---

48. Mini Project: Refactor a Messy Python Program

Start with:

```
def process(data):
    result = []

    for x in data:
        if x["active"] == True:
            if x["price"] > 0:
                if x["category"] != "deleted":
                    result.append(x["price"] * 0.9)

    return result
```

Your goal is to make it easier to understand.

Possible direction:

```
DISCOUNT_RATE = 0.10


def is_valid_product(product):
    return (
        product["active"]
        and product["price"] > 0
        and product["category"] != "deleted"
    )


def calculate_discounted_price(price):
    return price * (1 - DISCOUNT_RATE)
```

```
def get_discounted_prices(products):
    return [
        calculate_discounted_price(product["price"])
        for product in products
        if is_valid_product(product)
    ]
```

Then write tests for:

- active products
- inactive products
- zero price
- negative price
- deleted products
- discount calculation

The goal is not to blindly extract functions.

The goal is to make responsibilities obvious.

---

49. 5 Minute Challenge

Take one Python file you have written before.

Find:

1. one unclear variable name
2. one duplicated rule
3. one deeply nested condition
4. one function doing too many things
5. one unnecessary comment
6. one magic number

Refactor each.

Then run your tests.

Finally compare the code before and after.

Ask:

> Is the new version easier to understand without requiring more explanation?

---

50. Exercises

Exercise 1: Naming

Rewrite:

```
x = get_data()
y = process(x)
z = save(y)
```

using meaningful names.

Exercise 2: Guard Clauses

Refactor a function containing three nested `if` statements into guard clauses.

Exercise 3: Function Responsibility

Find a function that:

- reads input
- validates data
- performs business logic
- saves data

Split it into appropriate responsibilities.

Exercise 4: Remove Duplication

Find a business rule repeated in at least three places.

Extract it into one clear function or constant.

Exercise 5: Testing

Take one function that depends on global state.

Refactor it so dependencies are passed explicitly.

Write tests for the new version.

---

51. Mini Quiz

1. What is maintainability?

Answer: The degree to which code remains understandable, testable, and easy to modify safely over time.

2. Should every function be extremely short?

Answer: No. Functions should have clear responsibilities and reasonable complexity.

3. What should comments usually explain?

Answer: Why something is necessary or non-obvious, rather than simply repeating what the code does.

4. What is premature abstraction?

Answer: Creating complex reusable structures before there is enough real complexity or repetition to justify them.

5. Why is explicit dependency injection useful?

Answer: It makes dependencies visible and can make components easier to test and replace.

6. Why should global mutable state be minimized?

Answer: It can create hidden dependencies and unpredictable behavior.

7. What is refactoring?

Answer: Improving the internal structure of code without intentionally changing its external behavior.

---

52. Interview Questions

Beginner

1. What does maintainable code mean?
2. Why are meaningful names important?
3. What is the purpose of a function?
4. What is code duplication?
5. What are magic numbers?
6. Why are comments not a replacement for readable code?
7. What is refactoring?

Intermediate

8. What is the Single Responsibility Principle?
9. What is coupling?
10. What is cohesion?
11. Why can global state be dangerous?
12. What is dependency injection?
13. How does testing influence code design?
14. What is premature abstraction?
15. When should code be refactored?
16. How can you reduce deep nesting?

17. Why should business logic be separated from I/O?
18. What causes circular dependencies?

Practical

19. How would you refactor a 500-line Python function?
20. How would you make a database-dependent service easier to test?
21. How would you handle repeated business rules?
22. How would you decide whether to create a new module?
23. How would you review a pull request for maintainability?
24. How would you improve a Python codebase without changing its behavior?

---

53. Cheat Sheet

Clear names

```
total_price
active_users
resource_repository
```

instead of:

```
x
data
thing
```

Constants

```
MAX_RETRIES = 3
```

instead of:

```
if retries > 3:
```

Guard clauses

```
if not user:
    return
```

instead of unnecessary deep nesting.

Focused functions

```
validate()
calculate()
```

```
save()
notify()
```

instead of one giant function.

Explicit dependencies

```
service = ResourceService(repository)
```

instead of hiding dependencies inside the service.

Type hints

```
def calculate_total(prices: list[float]) -> float:
    ...
```

Meaningful comments

```
# API pagination starts at page 1.
```

Testing

```
Change
 ↓
Run tests
 ↓
Refactor
 ↓
Run tests again
```

Maintainability test

Ask:

```
Can I understand this?
Can I test this?
Can I change this safely?
```

Key Takeaways

- Maintainable Python is designed for the future, not only for today.
- Clear names communicate intent.
- Functions should have focused responsibilities.
- Avoid unnecessary nesting and use guard clauses when appropriate.
- Remove duplicated business knowledge.

- Do not overuse abstractions.
- Keep modules organized around meaningful responsibilities.
- Separate business logic from I/O and infrastructure where useful.
- Make dependencies explicit.
- Minimize global mutable state.
- Use type hints and documentation when they provide useful information.
- Comments should explain why, not restate the code.
- Remove dead code and obsolete comments.
- Refactor gradually and use tests to protect behavior.
- Formatting and linting improve consistency, but maintainability goes beyond style.
- Good Git history, tests, documentation, and project structure all contribute to maintainability.
- The simplest design that clearly handles the real problem is often easier to maintain than an elaborate architecture.

---

haas.dev

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

Visit haas.dev for more resources and guides.