Python Comments and Code Readability: Write Code Humans Can Understand

Learning Objectives

By the end of this guide, you will understand:

- What comments are and why developers use them
- How to write single line comments
- How comments differ from executable code
- How to document important decisions
- What makes Python code readable
- Python naming and spacing conventions
- How to organize related code
- How to avoid unnecessary comments
- How to write readable conditions and expressions
- How readability affects debugging and maintenance
- How professional developers make code easier for others to understand

---

1. Code Is Read by Humans Too

When you start programming, your main goal is usually:

> Make the code work.

That is important.

But professional software has another requirement:

> Make the code understandable.

Consider:

```
x = 120000
y = 2
z = x * y
```

Python can execute this.

But another developer has to guess:

- What is `x`?
- What is `y`?
- What does `z` represent?

Now compare it with:

```
product_price = 120000
quantity = 2
total_price = product_price * quantity
```

The second version communicates its purpose immediately.

Both programs can produce the same result.

The second one is more readable.

---

2. What Is Code Readability?

Code readability is how easily a developer can understand code by looking at it.

Readable code makes it easier to answer:

- What does this code do?
- Why does it exist?
- What information is being used?
- Where should I make a change?
- What happens next?
- Is there a potential problem?

Readable code is especially important when projects become large.

A program you understand today may be maintained by someone else months or years later.

---

3. Why Readability Matters

Imagine a real application such as an e-commerce platform.

Thousands of lines of code may handle:

- users
- products
- payments
- orders
- inventory
- notifications
- authentication
- shipping

If the code is difficult to understand, even a small change becomes risky.

Readable code helps developers:

- debug faster
- review code
- collaborate
- add features
- find bugs
- maintain old systems
- understand unfamiliar projects

Writing readable code is therefore an engineering skill, not just a formatting preference.

---

4. What Is a Comment?

A comment is text written in source code for developers.

Python ignores comments during normal execution.

A single line comment begins with:

```
#
```

Example:

```
# Display the welcome message
print("Welcome to Python")
```

The comment explains the code to a human.

Python executes:

```
print("Welcome to Python")
```

but does not execute the comment as a Python instruction.

---

5. Your First Comment

Example:

```
# Store the user's name
name = "Hafsa"

print(name)
```

The comment tells the reader what the next line is doing.

This can be useful when the purpose of the code is not immediately obvious.

---

6. Comments Can Appear Above Code

The most common style is to place a comment above the code it describes.

```
# Calculate the total order price
total_price = price * quantity
```

Another example:

```
# Check whether the user has administrator access
if is_admin:
    print("Admin dashboard")
```

This makes the relationship between the comment and code easy to understand.

---

7. Comments Can Also Appear Beside Code

Python allows inline comments.

```
age = 25  # User's current age
```

This can be useful for short explanations.

However, avoid filling every line with comments.

For example, this is unnecessary:

```
name = "Hafsa"  # Store name
age = 25        # Store age
```

The code is already obvious.

Prefer comments when they provide additional context.

---

8. Good Comments Explain Why

One of the most important rules of commenting is:

> Good comments often explain **why**, not simply **what**.

Consider:

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

The code already shows what happens.

This comment adds little value.

A better comment might explain an unusual reason:

```
# Start the count at 1 because the UI displays the first item as position 1
count = 1
```

The comment provides information that isn't obvious from the code itself.

---

9. Bad Comment vs Useful Comment

Less useful

```
# Multiply price by quantity
total = price * quantity
```

The code already tells us this.

More useful

```
# Inventory uses units, so calculate the order total before applying shipping
total = price * quantity
```

The second comment provides context.

---

10. Don't Use Comments to Explain Bad Code

Sometimes developers write complicated code and then add a long comment to explain it.

For example:

```
# Check whether the user has permission by comparing the role,
# checking the account status, making sure the subscription is active,
# and then determining whether the requested resource is available.
if u and r == "admin" and s and p:
    access = True
```

A better solution may be to make the code itself clearer.

```
is_admin = role == "admin"
has_active_subscription = subscription_active
has_permission = is_admin and has_active_subscription

if has_permission:
    access = True
```

Now the code communicates much of the meaning by itself.

A useful principle:

> Prefer clear code first. Use comments for information the code cannot communicate naturally.

---

11. Meaningful Variable Names

Variable names are one of the strongest readability tools.

Poor:

```
x = 5000
y = 2
z = x * y
```

Better:

```
product_price = 5000
quantity = 2
total_price = product_price * quantity
```

The second version reduces the need for comments.

---

12. Naming Boolean Variables

Boolean variables represent true/false information.

Compare:

```
x = True
```

with:

```
is_logged_in = True
```

The second immediately communicates what the Boolean means.

Other examples:

```
is_active = True
has_access = False
can_edit = True
should_retry = False
```

Good names make conditions easier to read.

---

13. Readable Conditions

Compare:

```
if u and a and s:
    print("Access granted")
```

with:

```
if is_logged_in and is_admin and has_access:
    print("Access granted")
```

The second version can almost be read like English.

Readable conditions reduce mental effort.

---

14. Use Consistent Naming

Python commonly uses `snake_case` for variables and functions.

Example:

```
first_name = "Hafsa"
last_name = "Waseem"
total_price = 10000
user_count = 50
```

Avoid inconsistent styles such as:

```
firstName = "Hafsa"
last_name = "Waseem"
TOTALprice = 10000
```

Consistency makes code easier to scan.

---

15. Avoid Extremely Short Names

Short names are sometimes useful for very small scopes.

For example:

```
for i in range(10):
    print(i)
```

`i` is commonly understandable as a loop counter.

But for important application data, avoid unexplained names like:

```
x = 50000
d = 10
p = x - d
```

Prefer:

```
account_balance = 50000
withdrawal = 10
remaining_balance = account_balance - withdrawal
```

The code now explains itself.

---

16. Avoid Extremely Long Names

Readable does not mean making every name enormous.

This is unnecessarily long:

```
total_price_of_all_products_in_the_current_shopping_cart = 5000
```

A clearer option might be:

```
cart_total = 5000
```

The goal is:

> Specific enough to understand, short enough to read comfortably.

---

17. Whitespace Improves Readability

Compare:

```
name="Hafsa"
age=25
total=100+50
```

with:

```
name = "Hafsa"
age = 25
total = 100 + 50
```

Both can represent valid Python.

The second is easier to read.

Use whitespace consistently around operators and assignments.

---

18. Blank Lines Help Group Related Code

Consider:

```
name = "Hafsa"
role = "Software Engineer"
language = "Python"
print(name)
print(role)
print(language)
```

You can visually separate data from output:

```
name = "Hafsa"
role = "Software Engineer"
language = "Python"

print(name)
print(role)
print(language)
```

The blank line communicates a change in purpose.

---

19. Group Related Code

Consider a simple application:

```
name = "Hafsa"
age = 25

price = 5000
quantity = 2

is_logged_in = True
```

The variables represent different concepts.

You can make the organization clearer:

```
# User information
name = "Hafsa"
age = 25

# Order information
price = 5000
quantity = 2

# Authentication state
is_logged_in = True
```

Now the sections are easier to scan.

---

20. Comments as Section Labels

Comments can help divide larger files into meaningful sections.

```
# User configuration
name = "Hafsa"
language = "Python"

# Application settings
debug_mode = True
max_items = 100

# Output
print(name)
print(language)
```

This can be useful when a file contains several logically different sections.

Do not turn every few lines into a giant heading system.

Use section comments when they genuinely improve navigation.

---

21. Keep Code Formatting Consistent

Imagine one section uses:

```
total_price = price * quantity
```

and another uses:

```
total=price*quantity
```

The difference may seem small.

But across thousands of lines, inconsistent formatting makes code harder to scan.

Consistent formatting gives the entire project a predictable visual structure.

Later, you will learn automatic formatting tools such as formatters.

---

22. One Logical Task at a Time

Avoid cramming unrelated operations into one complicated expression.

For example:

```
result = (price * quantity) - discount + shipping - refund
```

This may be fine if the calculation is genuinely simple.

But if the business logic becomes complicated, separate meaningful steps:

```
subtotal = price * quantity
discounted_total = subtotal - discount
total_with_shipping = discounted_total + shipping
final_total = total_with_shipping - refund
```

Now each step has a name.

This makes debugging easier.

---

23. Readability and Debugging

Readable code makes bugs easier to locate.

Compare:

```
x = a * b - c + d
```

with:

```
subtotal = price * quantity
discounted_total = subtotal - discount
final_total = discounted_total + shipping
```

If the final number is wrong, the second version gives you several meaningful points to inspect.

Readable code helps developers reason about the program.

---

24. Comments and Debugging

Comments can explain temporary debugging context.

For example:

```
# Temporary logging while investigating incorrect order totals
print("Order total:", total)
```

Once the investigation is finished, temporary debugging code should usually be removed or replaced with proper logging.

Do not leave large amounts of obsolete debugging comments throughout a production codebase.

---

25. Comments Should Stay Accurate

One of the worst things a comment can do is lie.

Example:

```
# Calculate the final price
price = 5000
```

Later, the code changes:

```
# Calculate the final price
discount = 500
```

If the comment no longer describes the code accurately, it becomes harmful.

Whenever code changes, check whether nearby comments still make sense.

---

26. Outdated Comments Are Dangerous

Imagine:

```
# Users can only upload images smaller than 2 MB
MAX_FILE_SIZE = 10
```

The code now allows 10 MB.

The comment says 2 MB.

A developer reading the file could make the wrong assumption.

The lesson:

> An incorrect comment can be worse than no comment.

Comments are part of the codebase and should be maintained.

---

27. Don't Comment Every Obvious Line

This:

```
# Create name
name = "Hafsa"

# Create age
age = 25

# Print name
print(name)

# Print age
print(age)
```

is unnecessarily verbose.

Better:

```
name = "Hafsa"
age = 25

print(name)
print(age)
```

The code is self-explanatory.

---

28. Comments Should Add Information

Useful comment:

```
# Keep this value at 60 because the external API rejects larger batch sizes.
batch_size = 60
```

The code alone cannot explain why `60` is special.

The comment preserves important context.

This is a strong use of comments.

---

29. Real World Example: Payment System

Imagine:

```
# The payment provider requires amounts in the smallest currency unit.
amount = price_in_rupees * 100
```

This comment explains a technical requirement.

Without it, another developer might "simplify" the code and accidentally break payments.

Comments are particularly valuable around:

- external APIs
- business rules
- security decisions
- unusual workarounds
- performance constraints
- compatibility requirements

---

30. Real World Example: haas.dev

Imagine the haas.dev resource system has a special rule:

```
# Keep PDF filenames lowercase because the deployment environment treats
paths as case-sensitive.
file_name = "python_variables.pdf"
```

The filename itself does not explain why lowercase matters.

The comment preserves the reason behind the decision.

That is a useful comment.

---

31. Comments and Business Rules

Suppose a learning platform gives users access to premium resources.

```
# Free resources are available without authentication.
if resource_is_free:
    print("Open resource")
```

The comment provides product context.

But if the code is already very clear:

```
if resource_is_free:
    print("Open resource")
```

the comment may not be necessary.

Always ask:

> Does this comment provide information the code cannot communicate?

---

32. Python Docstrings Are Different

You may encounter:

```
def calculate_total():
    """Calculate the final order total."""
```

This is a **docstring**, not simply a comment.

Docstrings are used to document functions, classes, and modules.

You will study them in detail later when you learn functions.

For now, remember the distinction:

```
# Comment
```

versus:

```
"""Docstring"""
```

Comments are primarily notes for developers.

Docstrings are structured documentation associated with Python objects.

---

33. Readable Functions

Even before learning functions deeply, you can understand the readability principle.

Less clear:

```
def x(a,b):
    return a*b
```

More descriptive:

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

The second version tells you:

- what the function does
- what the inputs represent
- what the result represents

Meaningful names reduce the need for explanation.

---

34. Keep Functions Focused

A function should ideally have a clear responsibility.

Instead of one huge function doing:

```
create user
validate payment
send email
generate report
update inventory
```

professional code often separates these responsibilities.

For example:

```
create_user()
process_payment()
send_confirmation_email()
generate_report()
update_inventory()
```

You will learn function design later.

The readability principle starts now:

> Give each piece of code a clear purpose.

---

35. Avoid Unnecessary Nesting

Compare:

```
if user:
    if user.is_logged_in:
        if user.has_access:
            print("Open resource")
```

with a design that can sometimes simplify the condition:

```
if user and user.is_logged_in and user.has_access:
    print("Open resource")
```

The exact choice depends on the application and whether intermediate conditions need separate handling.

The principle is not "always make code shorter."

The principle is:

> Make the structure easy to understand.

---

36. Readability Is Not the Same as Fewer Lines

Beginners sometimes think:

> Fewer lines = better code.

Not necessarily.

Compare:

```
final = p*q-d+s-r
```

with:

```
subtotal = product_price * quantity
discounted_total = subtotal - discount
total_with_shipping = discounted_total + shipping
final_total = total_with_shipping - refund
```

The second version uses more lines.

But it communicates the process much more clearly.

Readable code is about clarity, not line count.

---

37. Readability vs Cleverness

Python allows developers to write concise code.

But overly clever code can be difficult for others to understand.

For beginner and production code alike:

> Prefer clear solutions over clever tricks when the clever version reduces readability.

A future developer should be able to understand the code without reverse-engineering it.

---

38. Use Familiar Python Patterns

Python has established conventions.

Following common patterns makes your code easier for Python developers to understand.

Examples include:

```
user_name = "Hafsa"
```

instead of unusual naming styles.

And:

```
if is_logged_in:
    print("Welcome")
```

instead of unnecessarily complicated alternatives.

As you progress, you will learn the broader principles known as **Pythonic code**.

---

39. A Readability Review Checklist

Before considering a piece of code finished, ask:

Names

- Do variable names explain their purpose?
- Are Boolean names clear?
- Are names consistent?

Structure

- Are related lines grouped together?
- Is indentation consistent?

- Is the code unnecessarily nested?

Comments

- Does each comment provide useful information?
- Does it explain why when necessary?
- Is it still accurate?

Formatting

- Is spacing consistent?
- Are long expressions readable?
- Are blank lines used appropriately?

Complexity

- Can another developer understand the code without excessive mental effort?
- Are there unnecessary tricks or clever shortcuts?

---

40. Before and After: Developer Code

Before

```
x="Hafsa"
y=25
z=True

if z:
 print(x)
 if y>=18:
  print("Adult")
```

This may be difficult to read.

After

```
name = "Hafsa"
age = 25
is_active = True

if is_active:
    print(name)

    if age >= 18:
        print("Adult")
```

The second version has:

- meaningful names
- consistent spacing
- consistent indentation
- clearer structure

---

41. Before and After: Product System

Before

```
p=5000
q=2
d=500
s=200
t=p*q-d+s
print(t)
```

After

```
product_price = 5000
quantity = 2
discount = 500
shipping = 200

subtotal = product_price * quantity
total = subtotal - discount + shipping

print("Total:", total)
```

The second version is longer but significantly easier to understand.

---

42. Before and After: haas.dev

Before

```
x="Python Variables"
y="Python Development"
z=25
a=True
```

After

```
resource_title = "Python Variables"
category = "Python Development"
page_count = 25
is_free = True
```

The second version makes the data model much clearer.

---

43. A Readable haas.dev Example

```
# Resource information
resource_title = "Python Comments and Code Readability"
category = "Python Development"
page_count = 25
is_free = True

# Display resource information
print("Title:", resource_title)
print("Category:", category)
print("Pages:", page_count)
print("Free:", is_free)
```

Notice that the comments describe sections rather than explaining every obvious line.

---

44. A Readable Order Example

```
# Order information
product_name = "Laptop"
product_price = 120000
quantity = 2

# Calculate order total
subtotal = product_price * quantity

print("Product:", product_name)
print("Quantity:", quantity)
print("Subtotal:", subtotal)
```

The code is simple enough that only the section-level comments are needed.

---

45. Five Principles of Readable Python

Remember these five principles:

1. Name things clearly

```
total_price
```

is better than:

```
x
```

when the meaning matters.

2. Keep structure visible

Use proper indentation and spacing.

3. Explain the unusual

Comments should preserve information that isn't obvious.

4. Avoid unnecessary complexity

Simple code is usually easier to maintain.

5. Keep comments accurate

A stale comment can mislead future developers.

---

46. Practice Exercises

Easy

Exercise 1

Rewrite this code using meaningful variable names:

```
x = "Hafsa"
y = 25
z = "Python"
```

Exercise 2

Add useful comments to:

```
price = 5000
quantity = 2
total = price * quantity
```

Do not comment every obvious line.

Exercise 3

Improve the formatting:

```
name="Hafsa"
age=25
print(name,age)
```

---

47. Medium

Exercise 4

Rewrite this code for readability:

```
p=120000
q=2
s=2000
t=p*q+s
print(t)
```

Use meaningful names and separate logical steps.

Exercise 5

Create a readable haas.dev resource record using:

- title
- category
- page count
- free status

Add only useful comments.

Exercise 6

Write a developer profile using clear variable names and logical sections.

---

48. Hard

Exercise 7

Take a piece of code you previously wrote and perform a readability review.

Look for:

- unclear names
- inconsistent spacing
- unnecessary comments
- missing comments
- unnecessary nesting
- confusing expressions

Rewrite it.

Exercise 8

Create a small order-processing program and make it understandable to someone who has never seen your code before.

Use comments only where the reason or business rule is not obvious.

---

49. Five Minute Challenge

Create a **Readable Developer Dashboard**.

Your program should contain:

- developer information
- project information
- a Boolean status
- at least one calculation
- at least two logical sections
- meaningful variable names
- correct indentation
- two or three useful comments

Your goal is not just to make the program work.

Your goal is:

> Make another developer understand it within a few seconds.

---

50. Mini Quiz

Question 1

What is the main purpose of a comment?

Answer: To provide useful information or context for developers without being executed as normal program instructions.

Question 2

Which is more readable?

```
x = 5000
```

or:

```
product_price = 5000
```

Answer: `product_price = 5000`, when the value represents a product price.

Question 3

Should every line of code have a comment?

Answer: No. Comments should add useful information rather than explain obvious code.

Question 4

What should a useful comment often explain?

Answer: Why a particular decision, workaround, constraint, or unusual piece of code exists.

Question 5

Can an outdated comment be harmful?

Answer: Yes. It can mislead developers about what the code actually does.

Question 6

What naming style is commonly used for Python variables?

Answer: `snake_case`.

---

51. Interview Questions

What is code readability?

Code readability is the degree to which developers can easily understand the purpose, structure, and behavior of source code.

Why is readable code important?

It makes debugging, maintenance, collaboration, code review, and future development easier.

What makes a good comment?

A good comment provides useful context that is not obvious from the code, especially explaining why something is implemented in a particular way.

Why should you avoid excessive comments?

Excessive comments create noise and can make code harder to scan. Clear code should communicate obvious behavior itself.

What is better: a comment or a meaningful variable name?

Whenever possible, make the code self-explanatory through meaningful names. Use comments for additional context that the code itself cannot communicate.

Can comments become incorrect?

Yes. If code changes and comments are not updated, comments can become misleading.

What is snake_case?

A Python naming convention where multiple words are separated by underscores:

```
total_price = 5000
```

Is shorter code always more readable?

No. A few additional meaningful lines can make complex logic much easier to understand.

---

52. Cheat Sheet

Single line comment

```
# This explains something important
```

Comment above code

```
# Calculate the order total
total = price * quantity
```

Inline comment

```
batch_size = 60  # Required by the external API
```

Meaningful variable names

```
user_name = "Hafsa"
total_price = 5000
is_logged_in = True
```

Group related code

```
# User information
name = "Hafsa"
age = 25

# Application settings
debug_mode = True
```

Readable calculation

```
subtotal = product_price * quantity
total = subtotal - discount + shipping
```

Avoid unclear names

```
# Avoid when meaning is important
x = 5000
y = 2
z = x * y
```

Prefer

```
product_price = 5000
quantity = 2
total_price = product_price * quantity
```

---

53. Key Takeaways

Remember:

1. Code is written for humans as well as computers.
2. Readability makes software easier to understand and maintain.
3. Comments are notes for developers.
4. Use `#` for single-line comments.
5. Good comments often explain **why**, not obvious **what**.
6. Don't comment every line.
7. Clear variable names can eliminate unnecessary comments.
8. Use meaningful names such as `total_price` instead of unclear names such as `x`.
9. Use `snake_case` for Python variables.
10. Keep indentation and spacing consistent.
11. Group related code together.
12. Use blank lines to separate logical sections.
13. Break complicated logic into meaningful steps when necessary.
14. Avoid unnecessary cleverness.
15. Keep comments accurate and up to date.
16. Readability is about clarity, not simply fewer lines.
17. Good code should communicate as much of its intent as possible by itself.
18. Comments should provide additional context rather than repeat the code.

---

54. The Professional Mindset

A beginner often asks:

> Does my code work?

A growing developer asks:

> Does my code work, and can I understand it tomorrow?

A professional developer also considers:

> Can another developer safely understand, modify, test, and maintain this code?

That is the mindset you should develop from the beginning.

Your goal is not merely to write code Python can execute.

Your goal is to write code **humans can understand**.

---

55. What Comes Next?

You now know:

```
Python Program
      ↓
Variables
      ↓
Syntax
      ↓
Indentation
      ↓
Comments
      ↓
Readable Code
```

You have seen values such as:

```
name = "Hafsa"
age = 25
rating = 4.8
is_active = True
```

But we have only used these values at a surface level.

Now it is time to understand what they actually are.

Why is `"Hafsa"` different from `25`?

Why is `4.8` different from `25`?

Why is `True` not a string?

What can you do with each type?

The next guide will answer these questions:

Python Data Types: Understand the Values Your Programs Use

You will learn how Python represents text, integers, floating-point numbers, Booleans, and `None`, and how data types determine what operations your program can perform.
