Python Comparison and Logical Operators

Build Conditions and Make Decisions

Learning Objectives

By the end of this guide, you will understand:

- What comparison operators are
- How Python compares values
- All six comparison operators
- Boolean results
- Comparing numbers, strings, and variables
- What logical operators are
- How `and`, `or`, and `not` work
- Truth tables
- Combining multiple conditions
- Operator precedence in conditions
- Chained comparisons
- Short circuit evaluation
- Truthy and falsy values
- Common mistakes
- Real world business rules
- How conditions are used in applications
- How to write readable conditional expressions

---

1. Why Comparison and Logical Operators Matter

A program constantly needs to answer questions.

For example:

```
Is the user logged in?
Is the password correct?
Is the product in stock?
Is the user's age greater than 18?
Is the order large enough for free delivery?
Does the user have permission?
```

Computers cannot make these decisions without precise conditions.

Python uses **comparison operators** to compare values and **logical operators** to combine conditions.

For example:

```
age = 25

age >= 18
```

Python evaluates this as:

```
True
```

Now imagine a more realistic rule:

```
age >= 18 and has_account
```

The program can now evaluate multiple requirements together.

This is the foundation of:

- authentication
- authorization
- validation
- filtering
- search
- e-commerce
- banking
- dashboards
- recommendation systems
- APIs
- automation
- application workflows

---

2. What Is a Condition?

A **condition** is an expression that can be evaluated as true or false.

Example:

```
age >= 18
```

The result is:

```
True
```

Another example:

```
age < 18
```

The result is:

```
False
```

Conditions are therefore closely connected to Python's Boolean data type.

---

3. Boolean Values

Python has two Boolean values:

```
True
False
```

Notice that the first letter is capitalized.

Correct:

```
True
False
```

Incorrect:

```
true
false
```

Example:

```
is_logged_in = True
account_blocked = False
```

Boolean values are frequently used in conditions.

---

4. Comparison Operators

Python provides six primary comparison operators:

Operator	Meaning
`==`	Equal to
`!=`	Not equal to

`>`	Greater than
`<`	Less than
`>=`	Greater than or equal to
`<=`	Less than or equal to

Each comparison produces a Boolean result.

---

5. Equal To `==`

The `==` operator checks whether two values are equal.

```
age = 25

print(age == 25)
```

Output:

```
True
```

If:

```
age = 20
```

then:

```
print(age == 25)
```

produces:

```
False
```

---

6. `==` Compares Values

Consider:

```
name = "Hafsa"
```

```
print(name == "Hafsa")
```

Output:

```
True
```

The comparison asks:

> Does the value stored in `name` equal `"Hafsa"`?

This is different from assignment.

```
name = "Hafsa"
```

means:

> Store `"Hafsa"` in `name`.

While:

```
name == "Hafsa"
```

means:

> Compare the value of `name` with `"Hafsa"`.

---

7. Not Equal `!=`

The `!=` operator checks whether two values are different.

```
age = 25

print(age != 18)
```

Output:

```
True
```

Because 25 and 18 are different.

Example:

```
password = "python123"

if password != "":
    print("Password was provided")
```

---

8. Greater Than `>`

The `>` operator checks whether the left value is greater than the right value.

```
score = 85

print(score > 50)
```

Output:

```
True
```

But:

```
score = 40

print(score > 50)
```

produces:

```
False
```

---

9. Less Than `<`

The `<` operator checks whether the left value is smaller.

```
stock = 3

print(stock < 5)
```

Output:

```
True
```

This is useful for application rules.

```
if stock < 5:
    print("Low stock")
```

---

10. Greater Than or Equal `>=`

The `>=` operator checks two possibilities:

```
greater than
OR
equal to
```

Example:

```
score = 50
```

```
print(score >= 50)
```

Output:

```
True
```

Because 50 is equal to 50.

This is often useful for minimum requirements:

```
if score >= 50:
    print("Pass")
```

---

11. Less Than or Equal `<=`

The `<=` operator checks whether a value is smaller than or equal to another value.

```
age = 18

print(age <= 18)
```

Output:

```
True
```

It is useful when a maximum limit exists.

For example:

```
if quantity <= 10:
    print("Order accepted")
```

---

12. Comparison Operator Examples

```
a = 10
b = 5

print(a == b)
print(a != b)
print(a > b)
print(a < b)
print(a >= b)
print(a <= b)
```

Results:

```
False
True
True
False
True
False
```

---

13. Comparison Operators Return Booleans

Consider:

```
result = 10 > 5
```

The result is not:

```
10
```

or:

```
5
```

It is:

```
True
```

Therefore:

```
print(type(result))
```

produces:

```
<class 'bool'>
```

This is important because comparison expressions produce Boolean values that can be used by other parts of your program.

---

14. Using Comparisons in `if`

A comparison becomes especially useful inside a condition.

```
age = 25

if age >= 18:
    print("Access allowed")
```

The expression:

```
age >= 18
```

produces:

```
True
```

Therefore Python executes the indented block.

---

15. Comparison With Variables

You normally compare variables rather than hardcoded values.

```
minimum_age = 18
user_age = 25

if user_age >= minimum_age:
    print("Eligible")
```

This makes the program easier to modify.

If the minimum age changes:

```
minimum_age = 21
```

the condition automatically uses the new value.

---

16. Comparing Strings

Comparison operators also work with strings.

```
name = "Hafsa"

print(name == "Hafsa")
```

Output:

```
True
```

You can also check inequality:

```
if name != "Asim":
    print("Different name")
```

String comparison is based on the strings' values.

---

17. Case Sensitivity

String comparison is case-sensitive.

```
print("Python" == "python")
```

Output:

```
False
```

Because:

```
Python
python
```

are different strings.

If your application should treat them as equivalent, normalize the input first:

```
language = "PYTHON"

if language.lower() == "python":
    print("Python selected")
```

---

18. Comparing Input From Users

Remember:

```
input()
```

returns a string.

Example:

```
age = input("Enter your age: ")
```

Now `age` is a string.

Convert it before numerical comparison:

```
age = int(input("Enter your age: "))

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

The correct flow is:

```
User Input
    ↓
String
    ↓
```

```
Convert to integer
    ↓
Compare
    ↓
Boolean result
    ↓
Decision
```

---

19. Logical Operators

Comparison operators answer individual questions.

Logical operators allow you to combine those questions.

Python provides three main logical operators:

```
and
or
not
```

For example:

```
age >= 18 and has_account
```

This combines two conditions.

---

20. The `and` Operator

`and` returns a truthy result only when both conditions are true.

Example:

```
age = 25
has_account = True

if age >= 18 and has_account:
    print("Access granted")
```

Both conditions are true:

```
age >= 18       → True
has_account     → True
```

Therefore:

```
True and True → True
```

---

21. `and` Truth Table

Condition A	Condition B	A `and` B
True	True	True
True	False	False
False	True	False
False	False	False

The key idea:

> `and` requires both conditions to be true.

---

22. Real World `and` Example

Imagine a banking application.

A transfer should only be allowed when:

```
account is active
AND
balance is sufficient
```

Python:

```
account_active = True
balance = 5000
amount = 2000

if account_active and balance >= amount:
    print("Transfer allowed")
```

Both conditions must pass.

---

23. Multiple `and` Conditions

You can combine more than two conditions.

```
age = 25
has_account = True
account_active = True

if age >= 18 and has_account and account_active:
    print("Access granted")
```

All three must be true.

Think:

```
A AND B AND C
```

If even one is false, the overall condition does not pass.

---

24. The `or` Operator

`or` requires at least one condition to be true.

Example:

```
is_admin = False
is_manager = True

if is_admin or is_manager:
    print("Dashboard access granted")
```

The first condition is false.

The second is true.

Therefore:

```
False or True → True
```

---

25. `or` Truth Table

Condition A	**Condition B**	**A `or` B**
True	True	True
True	False	True
False	True	True

False	False	False

The key idea:

> `or` needs at least one true condition.

---

26. Real World `or` Example

Suppose a support system allows priority access to:

- premium users
- enterprise users

```
is_premium = False
is_enterprise = True

if is_premium or is_enterprise:
    print("Priority support")
```

Because one condition is true, access is granted.

---

27. The `not` Operator

`not` reverses a Boolean condition.

```
logged_in = False

if not logged_in:
    print("Please log in")
```

Because:

```
logged_in = False
```

then:

```
not logged_in = True
```

---

28. `not` Truth Table

Condition	**`not` Condition**

True	False
False	True

Think of `not` as:

> Reverse this Boolean result.

---

29. Real World `not` Example

```
account_blocked = False

if not account_blocked:
    print("Account can continue")
```

This reads almost like plain English:

> If the account is not blocked, continue.

Readable conditions are easier to maintain.

---

30. Combining `and`, `or`, and `not`

You can combine all three.

```
age = 25
has_account = True
blocked = False

if age >= 18 and has_account and not blocked:
    print("Access granted")
```

The program checks:

```
Age requirement
AND
Account exists
AND
Account is not blocked
```

---

31. Parentheses in Logical Conditions

Parentheses are extremely useful when combining `and` and `or`.

Consider:

```
if is_admin or is_manager and account_active:
    ...
```

Python's precedence rules determine how this is evaluated.

A clearer version is:

```
if is_admin or (is_manager and account_active):
    ...
```

Or, if your business rule means the account must be active regardless of role:

```
if (is_admin or is_manager) and account_active:
    ...
```

These are **different rules**.

Therefore, parentheses should be used to communicate the intended business logic.

---

32. A Practical Access Control Example

Suppose a dashboard should be accessible when:

```
user is an admin
OR
user is a manager

AND

account must be active
```

Write:

```
is_admin = False
is_manager = True
account_active = True

if (is_admin or is_manager) and account_active:
    print("Access granted")
else:
    print("Access denied")
```

The parentheses make the business rule obvious.

---

33. Comparison + Logical Operators

The most common real-world pattern is:

```
comparison + logical operator + comparison
```

Example:

```
age = 25
country = "Pakistan"

if age >= 18 and country == "Pakistan":
    print("Eligible")
```

There are two comparisons:

```
age >= 18
country == "Pakistan"
```

and one logical operator:

```
and
```

---

34. Chained Comparisons

Python allows elegant range checks.

Instead of:

```
age >= 18 and age <= 60
```

you can write:

```
18 <= age <= 60
```

Example:

```
age = 25

if 18 <= age <= 60:
    print("Age is within the allowed range")
```

This is called a **chained comparison**.

---

35. More Chained Comparison Examples

```
score = 75

if 0 <= score <= 100:
    print("Valid score")
```

Another:

```
temperature = 25

if 20 <= temperature <= 30:
    print("Comfortable range")
```

Chained comparisons are often easier to read than repeated conditions.

---

36. Multiple Comparisons

Python can also compare several values in a chain.

```
a = 10
b = 20
c = 30

print(a < b < c)
```

Output:

```
True
```

This represents:

```
a < b
AND
b < c
```

---

37. Membership Conditions

Comparison and logical operators frequently work with membership checks.

```
role = "admin"

if role in ["admin", "manager"]:
    print("Staff access")
```

You can combine this with `and`:

```
account_active = True

if role in ["admin", "manager"] and account_active:
    print("Staff access")
```

---

38. Identity Conditions

Identity checks can also be part of logical expressions.

A common example is:

```
value = None

if value is None:
    print("No value provided")
```

You can combine it:

```
if value is None or value == "":
    print("Missing value")
```

---

39. `==` vs `is`

This distinction deserves special attention.

`==`

Checks whether values are equal.

```
a = [1, 2]
b = [1, 2]

print(a == b)
```

Result:

```
True
```

`is`

Checks whether both references point to the same object.

```
print(a is b)
```

Result:

```
False
```

For ordinary value comparison, use:

```
==
```

For identity checks, use:

```
is
```

A common and recommended example is:

```
if value is None:
    ...
```

---

40. Truthy and Falsy Values

Python conditions do not always have to contain explicit:

```
True
```

or:

```
False
```

Many values can be interpreted as true or false in a Boolean context.

Examples of commonly falsy values:

```
False
None
0
0.0
""
[]
{}
set()
```

Many non-empty or non-zero values are truthy.

---

41. Truthy Example

Instead of:

```
name = "Hafsa"

if name != "":
    print("Name provided")
```

you can write:

```
if name:
    print("Name provided")
```

An empty string is falsy.

A non-empty string is truthy.

---

42. Falsy Example

```
items = []

if not items:
    print("No items available")
```

Because an empty list is falsy:

```
not [] → True
```

This pattern is common in Python.

---

43. Avoid Unnecessary Boolean Comparisons

Instead of:

```
if is_active == True:
    print("Active")
```

prefer:

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

Instead of:

```
if is_active == False:
    print("Inactive")
```

prefer:

```
if not is_active:
    print("Inactive")
```

This is cleaner Python.

---

44. Short Circuit Evaluation

Python does not always evaluate every part of a logical expression.

This is called **short circuit evaluation**.

For `and`:

```
False and anything
```

already has a false outcome.

For `or`:

```
True or anything
```

already has a true outcome.

Python can therefore stop evaluating once the result is known.

---

45. `and` Short Circuit Example

```
is_logged_in = False

if is_logged_in and check_permission():
    print("Access granted")
```

If `is_logged_in` is false, Python does not need to evaluate:

```
check_permission()
```

because:

```
False AND anything = False
```

This is both a performance feature and an important part of Python's behavior.

---

46. `or` Short Circuit Example

```
is_admin = True

if is_admin or check_special_permission():
    print("Access granted")
```

Because:

```
True OR anything = True
```

Python can stop after `is_admin`.

---

47. Why Short Circuiting Matters

Short circuiting can also prevent errors.

Example:

```
items = []

if items and items[0] == "Python":
    print("Found Python")
```

If `items` is empty, the second condition is not evaluated.

Without the short-circuit behavior, accessing:

```
items[0]
```

could raise an `IndexError`.

---

48. Comparing Numbers Safely

Suppose you receive a score from a user.

```
score = int(input("Enter score: "))

if 0 <= score <= 100:
    print("Valid score")
else:
    print("Invalid score")
```

The condition validates the complete range.

This is much safer than assuming every input is valid.

---

49. Combining Validation Rules

Suppose a username must:

- exist
- have at least 3 characters
- not be longer than 20 characters

You could write:

```
username = "Hafsa"

if username and 3 <= len(username) <= 20:
    print("Valid username")
```

Here:

```
username
AND
length >= 3
AND
length <= 20
```

are being evaluated together.

---

50. haas.dev Example

Imagine a haas.dev resource page.

A resource should be displayed only when:

```
resource exists
AND
resource is published
```

Example:

```
resource_exists = True
published = True

if resource_exists and published:
    print("Show resource")
```

A slightly larger rule could be:

```
resource_exists = True
published = True
is_free = True
user_logged_in = False

if resource_exists and published and (is_free or user_logged_in):
    print("Allow access")
```

The program is now expressing a real product rule.

---

51. E Commerce Example

Suppose an online store allows free shipping when:

```
order total >= 5000
OR
```

```
customer is premium
```

```
order_total = 3500
is_premium = True

if order_total >= 5000 or is_premium:
    shipping_fee = 0
else:
    shipping_fee = 250
```

The logical operator represents the business rule directly.

---

52. Authentication Example

A login system might require:

```
username correct
AND
password correct
AND
account active
```

```
username_correct = True
password_correct = True
account_active = True

if username_correct and password_correct and account_active:
    print("Login successful")
else:
    print("Login failed")
```

Notice that each condition represents a separate fact.

---

53. Permission Example

Suppose a user can delete a resource when they are either:

- an admin
- the resource owner

and their account is active.

```
is_admin = False
is_owner = True
account_active = True
```

```
if (is_admin or is_owner) and account_active:
    print("Delete allowed")
else:
    print("Delete denied")
```

This is a common authorization pattern.

---

54. Search Filtering Example

Imagine filtering products.

A product should be displayed when:

```
price <= user's maximum budget
AND
product is available
```

```
price = 3500
max_budget = 5000
available = True

if price <= max_budget and available:
    print("Show product")
```

This same logic can eventually become part of database queries, APIs, and search systems.

---

55. Nested Conditions vs Logical Operators

You can sometimes express the same rule using nested `if` statements.

Logical version

```
if age >= 18 and has_account:
    print("Access granted")
```

Nested version

```
if age >= 18:
    if has_account:
        print("Access granted")
```

Both can represent the same basic rule.

Use the form that makes the business logic easiest to understand.

---

56. When Conditions Become Too Complex

Avoid creating giant Boolean expressions.

For example:

```
if age >= 18 and account_active and not blocked and has_subscription and
(is_admin or is_owner) and country == "Pakistan":
    ...
```

It may technically work, but it becomes difficult to understand.

Instead, create meaningful intermediate variables:

```
adult_user = age >= 18
valid_account = account_active and not blocked
authorized = is_admin or is_owner
eligible_country = country == "Pakistan"

if adult_user and valid_account and has_subscription and authorized and
eligible_country:
    print("Access granted")
```

Now each business rule has a name.

---

57. Name Complex Conditions

This is one of the most useful habits in real Python projects.

Instead of:

```
if age >= 18 and has_account and not blocked:
    ...
```

you can write:

```
can_access = age >= 18 and has_account and not blocked

if can_access:
    print("Access granted")
```

This makes the code easier to:

- test
- debug
- read
- reuse

- modify

---

58. Operator Precedence in Conditions

Python evaluates operators according to precedence.

A simplified order relevant to conditions is:

```
()
not
and
or
```

For example:

```
A or B and C
```

is interpreted approximately as:

```
A or (B and C)
```

because `and` has higher precedence than `or`.

Still, when a condition represents important business logic, explicit parentheses are usually clearer.

---

59. Readability Beats Cleverness

This:

```
if (is_admin or is_owner) and account_active:
```

is easier to understand than:

```
if is_admin or is_owner and account_active:
```

Even when you know Python's precedence rules, parentheses communicate the intended rule immediately.

For production code:

> Make the condition easy for another developer to understand.

---

60. Common Mistake: Assignment Instead of Comparison

Incorrect:

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

Correct:

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

Remember:

```
=   → assignment
==  → equality comparison
```

---

61. Common Mistake: Forgetting Input Conversion

Incorrect:

```
age = input("Enter age: ")

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

`age` is a string.

Correct:

```
age = int(input("Enter age: "))

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

---

62. Common Mistake: Incorrect `or` Logic

Suppose you want to check whether a role is either `"admin"` or `"manager"`.

Incorrect:

```
if role == "admin" or "manager":
    print("Staff")
```

This does not mean what it appears to mean.

Correct:

```
if role == "admin" or role == "manager":
    print("Staff")
```

Or more cleanly:

```
if role in ["admin", "manager"]:
    print("Staff")
```

---

63. Common Mistake: Overusing `not`

A condition like:

```
if not not active:
```

is unnecessarily confusing.

Prefer:

```
if active:
```

Write conditions in the simplest form that communicates the rule.

---

64. Common Mistake: Confusing `and` With `or`

Suppose access requires:

```
admin OR manager
```

You need:

```
if is_admin or is_manager:
```

Using:

```
if is_admin and is_manager:
```

would require the user to be both.

Always translate the business rule into plain language before writing the condition.

---

65. Common Mistake: Ignoring Boundary Values

Suppose a discount applies to orders of 5000 or more.

This:

```
if total > 5000:
```

does not include exactly 5000.

Correct:

```
if total >= 5000:
```

Boundary values are important in:

- pricing
- age restrictions
- scores
- limits
- pagination
- validation
- quotas

---

66. Boundary Testing

For:

```
if score >= 50:
```

test:

```
49 → False
50 → True
51 → True
```

Do not only test obvious values.

For:

```
if quantity <= 10:
```

test:

```
9  → True
10 → True
11 → False
```

This habit prevents many bugs.

---

67. Practical Mini Project: Access Checker

Build a small access system.

```
age = 25
has_account = True
account_active = True

if age >= 18 and has_account and account_active:
    print("Access granted")
```

```
else:
    print("Access denied")
```

Now extend it:

```
is_admin = False

if (age >= 18 and has_account and account_active) or is_admin:
    print("Access granted")
else:
    print("Access denied")
```

You now have an exception rule:

> An admin can access the system even if the normal requirements are not all met.

Whether that is appropriate depends on the actual application's security rules.

---

68. Practical Mini Project: Shipping Calculator

Requirements:

```
Order >= 5000 → Free shipping
Premium customer → Free shipping
Otherwise → 250 shipping
```

Solution:

```
order_total = 4200
is_premium = False

if order_total >= 5000 or is_premium:
    shipping = 0
else:
    shipping = 250

print(f"Shipping: {shipping}")
```

---

69. Practical Mini Project: Result Validator

Requirements:

```
Score must be between 0 and 100.
Score >= 50 means pass.
```

```
score = 78

if 0 <= score <= 100:
    if score >= 50:
        print("Pass")
    else:
        print("Fail")
else:
    print("Invalid score")
```

This demonstrates why validation and business decisions are often separate stages.

---

70. Practical Mini Project: Product Filter

Requirements:

```
Product must:
- be available
- cost no more than budget
- have a rating of at least 4
```

```
price = 3500
budget = 5000
rating = 4.5
available = True

if available and price <= budget and rating >= 4:
    print("Product matches")
else:
    print("Product does not match")
```

---

71. Five Minute Challenge

Build an eligibility checker.

A user is eligible when:

```
age >= 18
AND
account is active
AND
user is either a premium member OR has a valid invitation
```

Create:

```
age
account_active
is_premium
has_invitation
```

Then write one condition that determines eligibility.

---

72. Practice Exercises

Easy

Exercise 1

Check whether:

```
number = 20
```

is greater than 10.

Exercise 2

Check whether:

```
age = 18
```

is greater than or equal to 18.

Exercise 3

Check whether two variables contain the same value.

Exercise 4

Check whether a number is not equal to zero.

Exercise 5

Create:

```
is_logged_in = False
```

and print a message when the user is not logged in.

---

73. Medium Exercises

Exercise 6

Check whether a score is between 50 and 100.

Use a chained comparison.

Exercise 7

Create a condition that requires:

```
age >= 18
AND
has_id == True
```

Exercise 8

Create a condition that allows access when:

```
is_admin
OR
is_manager
```

Exercise 9

Check whether a list contains `"Python"`.

Exercise 10

Check whether:

```
account_active
AND
not account_blocked
```

are both true.

---

74. Hard Exercises

Exercise 11: Discount Eligibility

A customer receives a discount when:

```
order >= 5000
OR
customer is premium
```

Implement the rule.

Exercise 12: Login Validation

A login is successful when:

```
username is correct
AND
password is correct
AND
account is active
```

Implement it using Boolean variables.

Exercise 13: Product Eligibility

A product can be purchased when:

```
price <= budget
AND
stock > 0
AND
product is active
```

Write the condition.

Exercise 14: Role Based Access

A user can access an admin dashboard when:

```
user is admin
OR
user is a manager with an active account
```

Write the condition with parentheses.

Exercise 15: Complete Validation

A username is valid when:

```
username is not empty
AND
length is between 3 and 20 characters
AND
username is not in a blocked list
```

Implement the validation.

---

75. Interview Questions

What are comparison operators?

Comparison operators compare values and produce Boolean results.

Examples:

```
==
!=
>
<
>=
<=
```

What does `==` do?

It checks whether two values are equal.

What is the difference between `=` and `==`?

`=` assigns a value.

`==` compares values.

What does `and` do?

It requires all relevant conditions to be true for the combined condition to be true.

What does `or` do?

It produces a truthy result when at least one relevant condition is true.

What does `not` do?

It reverses a Boolean condition.

What is short circuit evaluation?

It is Python's behavior of stopping evaluation of a logical expression when its final result is already determined.

What is a chained comparison?

A comparison such as:

```
18 <= age <= 60
```

which lets Python express a range check directly.

What is the difference between `==` and `is`?

`==` compares values.

`is` checks object identity.

Why are parentheses useful in logical expressions?

They make evaluation order and business rules explicit, especially when mixing `and` and `or`.

---

76. Comparison Cheat Sheet

```
a == b    # equal
a != b    # not equal
a > b     # greater than
a < b     # less than
a >= b    # greater than or equal
a <= b    # less than or equal
```

---

77. Logical Cheat Sheet

```
A and B   # both must be true
A or B    # at least one must be true
not A     # reverses A
```

---

78. Common Condition Patterns

Minimum requirement

```
score >= 50
```

Maximum limit

```
quantity <= 10
```

Range

```
10 <= number <= 100
```

Both conditions

```
condition_a and condition_b
```

Either condition

```
condition_a or condition_b
```

Negative condition

```
not condition
```

Membership

```
role in ["admin", "manager"]
```

Missing value

```
value is None
```

---

79. The Condition Building Framework

When writing a condition, use this process:

Step 1: Write the rule in plain English

Example:

```
User must be 18 or older and have an active account.
```

Step 2: Break it into facts

```
age >= 18
account_active
```

Step 3: Connect the facts

```
age >= 18 and account_active
```

Step 4: Add parentheses when useful

```
(age >= 18) and account_active
```

Step 5: Test boundary cases

```
17
18
19
```

This process is much more reliable than trying to write a complicated condition immediately.

---

80. From Conditions to Real Software

Comparison and logical operators become much more powerful when combined with other Python concepts.

A real application can follow:

```
User Input
     ↓
Validation
     ↓
Comparison
     ↓
Logical Conditions
     ↓
Business Rule
     ↓
Decision
     ↓
Application Action
```

For example:

```
age = int(input("Enter age: "))
has_account = True
account_active = True

eligible = (
    age >= 18
    and has_account
    and account_active
)

if eligible:
    print("Access granted")
else:
    print("Access denied")
```

This is the transition from learning syntax to writing actual program logic.

---

81. Key Takeaways

- Conditions are expressions that evaluate to Boolean results.
- Comparison operators compare values.
- `==` checks equality.

- `!=` checks inequality.
- `>` checks whether one value is greater.
- `<` checks whether one value is smaller.
- `>=` includes equality.
- `<=` includes equality.
- `and` requires all connected conditions to be true.
- `or` requires at least one condition to be true.
- `not` reverses a Boolean condition.
- Python supports chained comparisons.
- `==` compares values while `is` checks identity.
- Logical expressions use short circuit evaluation.
- Empty and zero-like values can be falsy.
- Parentheses make complex conditions easier to understand.
- Always consider boundary values when validating conditions.
- Meaningful Boolean variables can make complex rules much easier to read.
- Comparison and logical operators are the foundation of decision-making in Python.

Final Mental Model

Do not memorize operators as isolated symbols.

Think about them as a system for turning **facts into decisions**:

```
VALUES
  ↓
COMPARISON
  ↓
True / False
  ↓
LOGICAL OPERATORS
  ↓
COMBINED CONDITION
  ↓
BUSINESS RULE
  ↓
DECISION
  ↓
ACTION
```

For example:

```
(age >= 18 and has_account) and not blocked
```

can be understood as:

```
Is the user an adult?
        ↓
Does the user have an account?
        ↓
Is the account NOT blocked?
        ↓
All required conditions?
        ↓
Access decision
```

Once you can translate real-world rules into small Boolean conditions, you are no longer just writing Python syntax—you are beginning to express **program logic**.

---

haas.dev

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

Visit haas.dev for more resources and guides.