Python Nested Conditions

Learn how to make decisions inside decisions and build logic for real-world programs.

haas.dev

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

---

1. Why Do We Need Nested Conditions?

So far, you have learned how to make a program choose between different outcomes using:

```
if
elif
else
```

But real-world decisions are often more than one level deep.

Imagine an online learning platform.

You might need to ask:

```
Is the user logged in?
        ↓
     Yes
        ↓
Is the user enrolled?
        ↓
     Yes
        ↓
Is the resource available?
        ↓
     Yes
        ↓
Allow access
```

The second question only makes sense **after the first condition is satisfied**.

This is where **nested conditions** become useful.

A nested condition is simply:

> **A conditional statement placed inside another conditional statement.**

---

2. What Is a Nested Condition?

A nested condition is an `if`, `elif`, or `else` block that contains another conditional statement.

Basic example:

```
age = 20

if age >= 18:
    if age < 60:
        print("Adult")
```

The inner `if` exists inside the outer `if`.

That is why it is called **nested**.

Think of it like a box inside another box:

```
Outer decision
└── Inner decision
```

---

3. Basic Syntax

The general structure looks like this:

```
if outer_condition:
    if inner_condition:
        # code
```

You can also have `else`:

```
if outer_condition:
    if inner_condition:
        # code
    else:
        # code
else:
    # code
```

And you can have `elif` inside the nested structure:

```
if outer_condition:
    if condition1:
        # code
    elif condition2:
        # code
    else:
        # code
else:
    # code
```

The indentation tells Python which condition belongs inside which block.

---

4. The Most Important Mental Model

Do not think of nested conditions as:

> "I put an `if` inside another `if`."

Think of them as:

> **"I only need to ask the second question if the first question has a particular answer."**

For example:

```
Do you have an account?
        |
      Yes
        ↓
Are you logged in?
        |
      Yes
        ↓
Show dashboard
```

If the person doesn't have an account, there is no reason to check whether they are logged in.

This is the real purpose of nesting.

---

5. A Simple Example

```
age = 20

if age >= 18:
    print("You are an adult.")

    if age >= 21:
        print("You are 21 or older.")
```

For:

```
age = 20
```

Python first checks:

```
age >= 18
```

This is `True`.

Then Python enters the outer block.

It checks:

```
age >= 21
```

This is `False`.

Output:

```
You are an adult.
```

The inner condition was checked because the outer condition allowed Python to reach it.

---

6. Execution Flow

Consider:

```
age = 25

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

    if age >= 21:
        print("21 or older")
```

Execution:

```
Start
  ↓
age >= 18?
  ↓
Yes
  ↓
Print "Adult"
  ↓
age >= 21?
  ↓
Yes
  ↓
Print "21 or older"
  ↓
End
```

The outer condition controls whether Python enters the inner decision.

---

7. What Happens When the Outer Condition Is False?

Consider:

```
age = 15

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

    if age >= 21:
        print("21 or older")
```

The first condition:

```
age >= 18
```

is false.

Python never enters the outer block.

Therefore, it never evaluates:

```
age >= 21
```

Output:

```

```

Nothing is printed.

This is an important property of nested conditions:

> **If the outer condition is false, the nested code is skipped.**

---

8. Nested `if` With `else`

Consider:

```
age = 20

if age >= 18:
    if age >= 21:
        print("21 or older")
    else:
        print("18 to 20")
else:
    print("Under 18")
```

Possible results:

```
Age 16 → Under 18
Age 19 → 18 to 20
```

```
Age 25 → 21 or older
```

There are two levels of decisions.

Outer decision:

```
Is age at least 18?
```

Inner decision:

```
Is age at least 21?
```

---

9. Nested `if` + `elif` + `else`

An inner condition can have its own complete decision chain.

```
age = 25

if age >= 18:
    if age >= 60:
        print("Senior")
    elif age >= 30:
        print("Adult")
    else:
        print("Young adult")
else:
    print("Minor")
```

This gives:

```
Below 18 → Minor
18–29    → Young adult
30–59    → Adult
60+      → Senior
```

The outer condition first establishes that the person is an adult.

Then the inner conditions classify the adult.

---

10. Nested Conditions With User Input

Nested conditions become more practical when the program receives input.

Example:

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

```
if age >= 18:
    has_id = input("Do you have an ID? ")

    if has_id == "yes":
        print("Entry allowed")
    else:
        print("ID required")
else:
    print("You must be 18 or older")
```

Notice the design.

The program only asks:

```
Do you have an ID?
```

if:

```
age >= 18
```

is true.

That is a meaningful use of nesting.

---

11. Why This Is Better Than Asking Everything

Imagine someone is under 18.

There is no reason to ask them:

```
Do you have an ID?
Are you a member?
Do you have a ticket?
Is your account verified?
```

if the first rule already prevents access.

Nested decisions can help a program follow a logical sequence.

---

12. Real-World Example: Login

Imagine a simple login system.

Rules:

1. Check whether the account exists.
2. If it exists, check whether the password is correct.
3. If the password is correct, check whether the account is active.

4. If active, allow access.

You could model that with nested conditions:

```
account_exists = True
password_correct = True
account_active = True

if account_exists:
    if password_correct:
        if account_active:
            print("Login successful")
        else:
            print("Account is inactive")
    else:
        print("Incorrect password")
else:
    print("Account not found")
```

The logic follows the real process.

---

13. Execution of the Login Example

Suppose:

```
account_exists = True
password_correct = False
account_active = True
```

Python checks:

```
Account exists?
     ↓
    Yes
     ↓
Password correct?
     ↓
    No
     ↓
Incorrect password
```

It never needs to check:

```
account_active
```

because the password check already failed.

This is an example of **dependent decisions**.

---

14. Nested Conditions in haas.dev

Imagine haas.dev has resources that require an account.

```
has_account = True
is_logged_in = True
resource_available = True

if has_account:
    if is_logged_in:
        if resource_available:
            print("Open resource")
        else:
            print("Resource unavailable")
    else:
        print("Please log in")
else:
    print("Create an account first")
```

The decisions happen in a logical order:

```
Account?
   ↓
Logged in?
   ↓
Resource available?
   ↓
Open resource
```

This is similar to logic you may eventually encounter in real web applications.

---

15. Nested Conditions and Access Control

Access control often involves multiple requirements.

Suppose:

```
Admin:
    Can access everything

Student:
```

```
    Can access student resources

Guest:
    Can access public resources
```

You might start with:

```
user_type = "student"

if user_type == "admin":
    print("Admin access")
elif user_type == "student":
    print("Student access")
else:
    print("Guest access")
```

Now suppose student resources require enrollment.

```
user_type = "student"
is_enrolled = True

if user_type == "admin":
    print("Admin access")
elif user_type == "student":
    if is_enrolled:
        print("Student resources available")
    else:
        print("Enrollment required")
else:
    print("Public resources available")
```

The inner condition only applies to students.

---

16. Nested Conditions Are About Dependency

This is the key idea.

Suppose you have:

```
Condition A
```

and:

```
Condition B
```

If B only matters when A is true, nesting can represent that dependency.

Example:

```
Is the user a student?
        ↓
       Yes
        ↓
Is the student enrolled?
```

But if B is independent of A, nesting may not be appropriate.

---

17. Nested vs Combined Conditions

Sometimes you can write nested conditions in a single condition.

Nested:

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

Combined:

```
if age >= 18 and has_ticket:
    print("Entry allowed")
```

Both can produce the same result.

So which should you use?

The answer depends on what you are trying to communicate.

---

18. When Combining Conditions Is Better

If the rule is simply:

> "The user must be 18 or older AND have a ticket."

Then this is often clearer:

```
if age >= 18 and has_ticket:
    print("Entry allowed")
```

There is no need to create an extra level of nesting.

---

19. When Nesting Is Better

Suppose the program needs to provide different messages:

```
Under 18
18+ but no ticket
18+ with ticket
```

Then nesting can make the decision path explicit:

```
if age >= 18:
    if has_ticket:
        print("Entry allowed")
    else:
        print("Ticket required")
else:
    print("You must be 18 or older")
```

Each failure has a different explanation.

---

20. Another Comparison

Nested

```
if logged_in:
    if is_admin:
        print("Admin dashboard")
```

Combined

```
if logged_in and is_admin:
    print("Admin dashboard")
```

If all you care about is whether both conditions are true, the combined version is often simpler.

If you need different actions depending on the first decision, nesting may communicate the logic better.

---

21. Nested Conditions With `or`

You can use logical operators inside nested conditions.

```
if has_account:
    if is_verified or is_admin:
        print("Access allowed")
    else:
        print("Verification required")
```

```
else:
    print("Create an account")
```

The inner condition means:

```
Verified OR Admin
```

At least one must be true.

---

22. Nested Conditions With `not`

You can also use `not`.

```
if has_account:
    if not is_banned:
        print("Account can access the platform")
    else:
        print("Account is blocked")
else:
    print("Account does not exist")
```

`not` reverses a Boolean value.

```
not True  → False
not False → True
```

---

23. Nested Conditions With `elif`

You can combine an outer `if` with an inner `elif`.

```
user_type = "student"
age = 20

if user_type == "student":
    if age < 18:
        print("Minor student")
    elif age < 25:
        print("Young student")
    else:
        print("Adult student")
else:
    print("Not a student")
```

The inner `elif` only matters when:

```
user_type == "student"
```

is true.

---

24. Multiple Levels of Nesting

Python technically allows many levels:

```
if condition1:
    if condition2:
        if condition3:
            if condition4:
                print("Something")
```

But just because Python allows it does not mean you should do it.

Deep nesting can make code difficult to:

- read
- debug
- modify
- test
- understand

A developer should always ask:

> **Can this logic be made simpler?**

---

25. The Problem With Deep Nesting

Consider:

```
if account_exists:
    if logged_in:
        if verified:
            if subscription_active:
                if resource_available:
                    print("Open resource")
```

This works.

But understanding the requirements now requires tracking five levels.

It becomes harder to determine:

- Why access failed
- Which condition blocked the user

- Where to add another rule
- How to test every possibility

This is a warning sign.

---

26. Simplifying Deep Nesting

You can sometimes combine conditions:

```
if (
    account_exists
    and logged_in
    and verified
    and subscription_active
    and resource_available
):
    print("Open resource")
```

This reduces nesting.

However, if each failed condition needs a different message, a different design may be better.

For example:

```
if not account_exists:
    print("Create an account")
elif not logged_in:
    print("Please log in")
elif not verified:
    print("Verify your account")
elif not subscription_active:
    print("Activate your subscription")
elif not resource_available:
    print("Resource unavailable")
else:
    print("Open resource")
```

This can be much easier to maintain.

---

27. Guard Conditions

Another useful technique is to handle failure cases early.

Instead of deeply nesting:

```
if account_exists:
    if logged_in:
        if verified:
            print("Open dashboard")
```

you can write:

```
if not account_exists:
    print("Create an account")
elif not logged_in:
    print("Please log in")
elif not verified:
    print("Verify your account")
else:
    print("Open dashboard")
```

This style keeps the main successful path closer to the bottom and avoids unnecessary indentation.

Later, you will encounter this idea in larger programs and functions.

---

28. Nested Conditions and Input Validation

Suppose you ask for a number.

```
number = int(input("Enter a number: "))

if number >= 0:
    if number % 2 == 0:
        print("Positive even number")
    else:
        print("Positive odd number")
else:
    print("Negative number")
```

There are two decisions:

Outer decision

```
Is the number non-negative?
```

Inner decision

```
Is it even?
```

The inner decision is relevant only to non-negative numbers.

---

29. Nested Conditions With Three Outcomes

You can build more detailed logic:

```
number = int(input("Enter a number: "))

if number >= 0:
    if number == 0:
        print("Zero")
    elif number % 2 == 0:
        print("Positive even")
    else:
        print("Positive odd")
else:
    print("Negative")
```

This demonstrates:

- outer `if`
- inner `if`
- inner `elif`
- inner `else`
- logical dependency

---

30. Real-World Example: Shopping Checkout

Imagine an online store.

Rules:

1. The cart must contain items.
2. The user must be logged in.
3. The payment method must be available.
4. Then checkout is allowed.

A simple nested version:

```
cart_has_items = True
logged_in = True
payment_available = True

if cart_has_items:
    if logged_in:
        if payment_available:
```

```
            print("Proceed to checkout")
        else:
            print("Payment unavailable")
    else:
        print("Please log in")
else:
    print("Your cart is empty")
```

This follows a dependency chain.

---

31. Real-World Example: Exam Eligibility

Suppose a university has this rule:

> A student can sit for the final exam if they are registered. If registered, attendance must also be at least 75%.

```
registered = True
attendance = 80

if registered:
    if attendance >= 75:
        print("Eligible for final exam")
    else:
        print("Attendance requirement not met")
else:
    print("Student is not registered")
```

The attendance condition only matters if registration exists.

---

32. Real-World Example: ATM

Imagine a simplified ATM.

Rules:

1. Check whether the card is valid.
2. If valid, check PIN.
3. If PIN is correct, check balance.
4. If balance is sufficient, allow withdrawal.

```
card_valid = True
pin_correct = True
balance = 5000
```

```
withdrawal = 2000

if card_valid:
    if pin_correct:
        if withdrawal <= balance:
            print("Withdrawal successful")
        else:
            print("Insufficient balance")
    else:
        print("Incorrect PIN")
else:
    print("Invalid card")
```

This is a classic example of sequential dependent decisions.

---

33. Real-World Example: Online Course Access

Suppose a student wants to open a premium Python PDF.

Rules:

```
Is the user logged in?
       ↓
Is the user enrolled?
       ↓
Is the resource premium?
       ↓
Does the user have premium access?
```

A simplified model:

```
logged_in = True
enrolled = True
resource_premium = True
has_premium = False

if logged_in:
    if enrolled:
        if resource_premium:
            if has_premium:
                print("Open premium resource")
            else:
                print("Premium access required")
        else:
```

```
            print("Open resource")
    else:
        print("Enrollment required")
else:
    print("Please log in")
```

This example demonstrates how nested conditions can model a real access workflow.

---

34. Nested Conditions and State

Many applications work with states.

For example:

```
status = "active"
role = "student"

if status == "active":
    if role == "admin":
        print("Admin dashboard")
    elif role == "student":
        print("Student dashboard")
    else:
        print("User dashboard")
else:
    print("Account inactive")
```

The outer condition checks account state.

The inner condition checks user role.

This is a useful way to think about complex application logic:

> **First determine the larger state, then determine the appropriate behavior inside that state.**

---

35. Nested Conditions and Program Structure

When you see:

```
if A:
    if B:
        do_something()
```

read it as:

> "If A is true, and once we are in that situation, check whether B is true."

This is different from simply reading:

```
if A and B:
```

The result can be similar, but the structure communicates the decision process differently.

---

36. Common Mistake: Incorrect Indentation

Incorrect:

```
if age >= 18:
if has_ticket:
    print("Allowed")
```

Python will produce an error.

Correct:

```
if age >= 18:
    if has_ticket:
        print("Allowed")
```

Every nested block needs deeper indentation.

---

37. Common Mistake: Misunderstanding Which `else` Belongs to Which `if`

Consider:

```
if age >= 18:
    if has_ticket:
        print("Allowed")
    else:
        print("Ticket required")
else:
    print("Under 18")
```

The first `else` belongs to:

```
if has_ticket:
```

The second `else` belongs to:

```
if age >= 18:
```

Indentation determines this relationship.

---

38. Common Mistake: Excessive Nesting

This:

```
if a:
    if b:
        if c:
            if d:
                print("Done")
```

may work, but it can become difficult to understand.

Always ask:

```
Can I combine these?
Can I reverse the conditions?
Can I use elif?
Can I return early?
Can I separate the logic into functions later?
```

You will learn functions soon enough to make this even cleaner.

---

39. Common Mistake: Nesting When `and` Is Enough

Instead of:

```
if age >= 18:
    if has_ticket:
        print("Allowed")
```

you may simply write:

```
if age >= 18 and has_ticket:
    print("Allowed")
```

If both conditions are just requirements for the same action, the combined version may be clearer.

---

40. Common Mistake: Not Thinking About Every Path

When conditions become nested, there can be many possible outcomes.

For:

```
if account_exists:
    if logged_in:
```

```
        if verified:
            print("Access")
```

ask:

```
What if account doesn't exist?
What if account exists but user isn't logged in?
What if logged in but not verified?
What if everything is correct?
```

A robust program should deliberately decide what happens in each relevant situation.

---

41. Decision Tree Thinking

Before coding complex nested conditions, draw a decision tree.

Example:

```
Account exists?
├── No → Create account
└── Yes
    └── Logged in?
        ├── No → Login
        └── Yes
            └── Verified?
                ├── No → Verify
                └── Yes → Open dashboard
```

Then translate the tree into Python.

This makes complex requirements easier to understand.

---

42. From Decision Tree to Python

Decision tree:

```
Account exists?
    No → Create account
    Yes → Logged in?
             No → Login
             Yes → Verified?
                      No → Verify
                      Yes → Dashboard
```

Python:

```
if not account_exists:
    print("Create account")
else:
    if not logged_in:
        print("Login")
    else:
        if not verified:
            print("Verify your account")
        else:
            print("Open dashboard")
```

This is a direct translation of the decision tree.

---

43. Improving the Same Logic

The same logic can often be written more simply:

```
if not account_exists:
    print("Create account")
elif not logged_in:
    print("Login")
elif not verified:
    print("Verify your account")
else:
    print("Open dashboard")
```

Both represent the same general decision flow.

The second version is flatter and often easier to maintain.

This teaches an important lesson:

> **Nested conditions are a tool, not a requirement.**

Use them when they make the logic clearer.

---

44. When Nested Conditions Make Sense

Nested conditions are especially useful when:

1. One decision depends on another

```
If account exists → check login
```

2. You have different logic inside a particular state

```
If user is a student → determine student access
```

3. You want to ask for additional information only when necessary

```
If age is valid → ask for ticket
```

4. You are modeling a decision tree

```
A → B → C → result
```

---

45. When Nested Conditions Should Be Avoided

Avoid unnecessary nesting when:

- `and` or `or` expresses the rule clearly
- the code becomes deeply indented
- the same conditions are repeated
- failure cases can be handled earlier
- the logic becomes difficult to test
- the nesting hides the main purpose of the code

Readable code is usually easier to debug and maintain.

---

46. Mini Project: Student Portal Access

Build a program with these rules:

```
If the student has an account:
    Check whether they are logged in.

If logged in:
    Check whether they are enrolled.

If enrolled:
    Open the student portal.

Otherwise:
    Show enrollment message.

If not logged in:
    Show login message.

If no account:
    Show registration message.
```

Start with:

```
has_account = True
logged_in = True
enrolled = False
```

Then implement the logic.

One possible solution:

```
if has_account:
    if logged_in:
        if enrolled:
            print("Student portal opened")
        else:
            print("Enrollment required")
    else:
        print("Please log in")
else:
    print("Please create an account")
```

---

47. Mini Project: Secure ATM

Create a program with:

```
Card valid?
    ↓
PIN correct?
    ↓
Withdrawal amount valid?
    ↓
Enough balance?
    ↓
Withdraw
```

Starter variables:

```
card_valid = True
pin_correct = True
balance = 5000
amount = 1000
```

Try to implement the decision tree using nested conditions.

Then improve it by adding meaningful error messages.

---

48. Mini Project: haas.dev Resource Access

Create a simplified resource access checker.

Variables:

```
logged_in = True
is_student = True
is_premium = False
resource_premium = True
```

Rules:

```
Not logged in → Login required

Logged in:
    Student?
        No → Public access
        Yes → Check resource

Premium resource?
        No → Open resource
        Yes → Check premium access

Premium access?
        Yes → Open resource
        No → Premium required
```

This is an excellent exercise because it resembles the kind of decision logic used in real applications.

---

49. Debugging Nested Conditions

When nested code doesn't behave correctly, don't stare at the whole program at once.

Check each decision individually.

For:

```
if has_account:
    if logged_in:
        if verified:
            print("Access")
```

debug using:

```
print(has_account)
print(logged_in)
print(verified)
```

You can also print checkpoints:

```
if has_account:
    print("Account exists")

    if logged_in:
        print("User logged in")

        if verified:
            print("User verified")
```

This lets you see how far the program gets.

---

50. Trace the Program Manually

Suppose:

```
has_account = True
logged_in = False
verified = True
```

Code:

```
if has_account:
    if logged_in:
        if verified:
            print("Access")
```

Trace:

```
has_account
→ True

logged_in
→ False

verified
→ never checked
```

Why?

Because Python never enters the third level.

This kind of manual tracing is an important debugging skill.

---

51. Testing Nested Conditions

Nested logic should be tested with different combinations.

For:

```
if has_account:
    if logged_in:
        if verified:
            print("Access")
```

test:

```
True, True, True
True, True, False
True, False, True
False, True, True
```

This exposes different paths through the program.

As programs become larger, systematic testing becomes increasingly important.

---

52. Truth Tables and Nested Logic

For two Boolean conditions:

```
A = has_account
B = logged_in
```

Possible combinations:

A	**B**	**What happens?**
False	False	Account required
False	True	Account required
True	False	Login required

True	True	Continue

Notice that the value of `logged_in` doesn't matter when there is no account.

That is exactly what the nested structure communicates.

---

53. Nested Conditions vs `elif`

These structures solve different problems.

Nested:

```
if account_exists:
    if logged_in:
        print("Dashboard")
```

Meaning:

> If account exists, then check another condition.

`elif`:

```
if account_type == "admin":
    print("Admin")
elif account_type == "student":
    print("Student")
```

Meaning:

> Choose one option from several alternatives.

You can also combine both:

```
if account_exists:
    if account_type == "admin":
        print("Admin")
    elif account_type == "student":
        print("Student")
else:
    print("Create account")
```

---

54. Nested Conditions vs `and`

Nested:

```
if logged_in:
    if verified:
        print("Access")
```

Combined:

```
if logged_in and verified:
    print("Access")
```

Use the form that best communicates the actual business rule.

If the inner condition is simply another requirement for the same action, `and` may be cleaner.

If the inner decision has its own meaningful branches, nesting can be useful.

---

55. Engineering Principle: Model the Decision

Don't start with syntax.

Start with the requirement.

Requirement:

> "Only logged-in students can access the course."

Translate:

```
Logged in?
   ↓
Student?
   ↓
Access
```

Then code:

```
if logged_in:
    if is_student:
        print("Course access granted")
```

Or, if no separate branches are needed:

```
if logged_in and is_student:
    print("Course access granted")
```

The important skill is deciding what the program actually needs to check.

---

56. Engineering Principle: Keep the Logic Understandable

Imagine another developer opens your code six months later.

They should be able to understand:

```
if has_account:
    if logged_in:
        print("Open dashboard")
```

without reverse-engineering a complicated expression.

Code is not only written for computers.

It is also written for humans.

---

57. Mini Quiz

Question 1

What is a nested condition?

A conditional statement placed inside another conditional statement.

Example:

```
if condition:
    if another_condition:
        print("Yes")
```

Question 2

What happens to the inner `if` when the outer `if` is false?

The inner condition is skipped because Python never enters the outer block.

Question 3

Can this:

```
if age >= 18:
    if has_ticket:
        print("Allowed")
```

be written using `and`?

Yes:

```
if age >= 18 and has_ticket:
    print("Allowed")
```

---

58. Five-Minute Challenge

Build a **Movie Ticket Eligibility Checker**.

Ask the user for:

```
Age
Has membership?
```

Rules:

```
Under 13 → Child ticket

13–17:
    If member → Discounted teen ticket
    Otherwise → Regular teen ticket

18+:
    If member → Discounted adult ticket
    Otherwise → Regular adult ticket
```

Use nested conditions.

Example:

```
Enter age: 16
Are you a member? yes

Discounted teen ticket
```

---

59. Challenge: Build a Login Flow

Create:

```
account_exists
password_correct
is_verified
```

The program should produce:

```
No account
Incorrect password
Verification required
Login successful
```

Try to build it first using nested conditions.

Then rewrite it using a flatter `if` / `elif` structure.

Compare which version is easier to understand.

---

60. Interview Questions

What is nested `if` in Python?

It is an `if` statement placed inside another conditional block.

Why are nested conditions useful?

They represent decisions where one condition determines whether another decision needs to be evaluated.

Can nested conditions contain `elif`?

Yes.

```
if outer:
    if condition1:
        ...
    elif condition2:
        ...
```

Can nested conditions contain `else`?

Yes.

Is nesting always better than using `and`?

No. If two conditions are simply requirements for the same action, `and` may produce clearer code.

What is a major disadvantage of excessive nesting?

Deep nesting can reduce readability and make code harder to maintain and test.

Does Python require a fixed indentation level?

Python requires consistent indentation to define blocks. Four spaces per level is the conventional style.

---

61. Cheat Sheet

Basic nested condition

```
if condition:
    if another_condition:
        print("Something")
```

Nested with `else`

```
if condition:
    if another_condition:
        print("A")
    else:
        print("B")
else:
    print("C")
```

Nested with `elif`

```
if condition:
    if condition1:
        print("A")
    elif condition2:
        print("B")
    else:
        print("C")
```

Combined alternative

Instead of:

```
if a:
    if b:
        print("Yes")
```

you may write:

```
if a and b:
    print("Yes")
```

Logical operators

```
and
or
not
```

---

62. Key Takeaways

After this lesson, you should understand:

- A nested condition is a condition inside another condition.
- The inner condition is evaluated only when Python reaches it.
- If the outer condition is false, the inner block is skipped.

- Nested conditions are useful for dependent decisions.
- `if`, `elif`, and `else` can all appear inside nested structures.
- Indentation defines the relationship between conditions.
- `and` can sometimes replace simple nesting.
- Excessive nesting can make code difficult to maintain.
- Decision trees can help you design complex conditional logic.
- Testing different combinations is important when conditions become complex.
- Real applications use nested decision logic for authentication, access control, payments, checkout, eligibility, and application states.
- The goal is not to create the most nested code.
- The goal is to create **clear, correct, maintainable decision logic**.

---

63. Final Practice Task

Build a **haas.dev Learning Access System**.

Your program should have:

```
has_account
logged_in
is_student
resource_available
is_premium
has_premium_access
```

Design the logic yourself.

Your program should handle at least these situations:

```
No account
Not logged in
Not a student
Resource unavailable
Free resource
Premium resource with access
Premium resource without access
```

Start by drawing the decision tree.

Then write the Python code.

Finally, test every important path.

The goal is not just to make the program work once.

The goal is to understand **why each condition exists, which conditions depend on others, and when nesting makes the code clearer than a long collection of unrelated conditions.**

---

3 Questions Before You Touch the Keyboard

1. Does this decision depend on another decision?
2. Can I express this more clearly with `and`, `or`, or `elif`?
3. What should happen on every possible path?

Visit haas.dev for more resources and guides.

haas.dev

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