Python `match...case`

Learn how Python matches values against different patterns and when `match...case` is useful for clean decision making.

haas.dev

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

---

1. Why Does Python Need `match...case`?

You have already learned how to make decisions with:

```
if
elif
else
```

For example:

```
choice = "login"

if choice == "login":
    print("Opening login page")
elif choice == "register":
    print("Opening registration page")
elif choice == "profile":
    print("Opening profile")
else:
    print("Unknown option")
```

This works.

But notice what we are repeatedly doing:

```
choice == "login"
choice == "register"
choice == "profile"
```

We are essentially asking:

> "Which value does this variable match?"

Python provides another tool for this type of problem:

```
match...case
```

It allows you to describe different possible patterns or values in a structured way.

---

2. What Is `match...case`?

`match...case` is Python's pattern matching syntax.

It was introduced in **Python 3.10**.

The basic structure is:

```
match value:
    case pattern1:
        # code
    case pattern2:
        # code
    case pattern3:
        # code
```

For example:

```
command = "login"

match command:
    case "login":
        print("Opening login page")
    case "register":
        print("Opening registration page")
    case "profile":
        print("Opening profile")
```

Python compares `command` with each case.

When a case matches, its block executes.

---

3. The Mental Model

Think of `match...case` as a set of labeled boxes.

Suppose:

```
status = "pending"
```

Python asks:

```
Does status match "success"?
        ↓ No

Does status match "pending"?
        ↓ Yes
```

```
Run this case
```

So:

```
match status:
    case "success":
        print("Completed")
    case "pending":
        print("Waiting")
```

produces:

```
Waiting
```

The important idea is:

> **`match` looks at a value, and `case` describes what that value should match.**

---

4. Basic Syntax

The general syntax is:

```
match expression:
    case pattern:
        statement
    case pattern:
        statement
    case pattern:
        statement
```

Notice the colon after:

```
match expression:
```

and after every:

```
case pattern:
```

Indentation is also required.

---

5. A Simple Example

```
day = "Monday"

match day:
```

```
    case "Monday":
        print("Start of the week")
    case "Tuesday":
        print("Second day")
    case "Wednesday":
        print("Middle of the week")
    case "Thursday":
        print("Almost there")
    case "Friday":
        print("Last working day")
    case "Saturday":
        print("Weekend")
    case "Sunday":
        print("Weekend")
```

For:

```
day = "Monday"
```

output:

```
Start of the week
```

---

6. `match` Checks Cases in Order

Consider:

```
number = 2

match number:
    case 1:
        print("One")
    case 2:
        print("Two")
    case 3:
        print("Three")
```

Python checks the cases from top to bottom.

```
1 → No
2 → Yes
```

It executes:

```
print("Two")
```

and finishes the match statement.

---

7. The `case _` Default Case

What if none of the cases match?

You can use:

```
case _:
```

The underscore acts as a wildcard.

Example:

```
command = "logout"

match command:
    case "login":
        print("Login")
    case "register":
        print("Register")
    case "profile":
        print("Profile")
    case _:
        print("Unknown command")
```

Output:

```
Unknown command
```

Think of:

```
case _:
```

as:

> **"Anything else."**

It is similar to the role of `else`.

---

8. `match...case` vs `if...elif...else`

The same logic can often be written both ways.

Using `if...elif`

```
command = "login"
```

```
if command == "login":
    print("Login")
elif command == "register":
    print("Register")
elif command == "profile":
    print("Profile")
else:
    print("Unknown command")
```

Using `match...case`

```
command = "login"

match command:
    case "login":
        print("Login")
    case "register":
        print("Register")
    case "profile":
        print("Profile")
    case _:
        print("Unknown command")
```

Both can work.

The difference is mainly in how the decision is expressed.

With `if`, you explicitly write comparisons.

With `match`, you describe patterns that the value should match.

---

9. When `match...case` Is Useful

`match...case` is especially useful when you have:

- commands
- menu options
- application states
- status values
- event types
- structured data
- multiple fixed patterns
- data with different shapes

For example:

```
login
logout
register
profile
settings
help
```

These are naturally represented as different cases.

---

10. Matching Strings

String matching is one of the easiest ways to understand `match`.

```
action = "download"

match action:
    case "view":
        print("Viewing resource")
    case "download":
        print("Downloading resource")
    case "share":
        print("Sharing resource")
    case "delete":
        print("Deleting resource")
    case _:
        print("Unknown action")
```

Output:

```
Downloading resource
```

---

11. Matching Numbers

`match` can also match integers.

```
option = 2

match option:
    case 1:
        print("Home")
    case 2:
        print("Resources")
    case 3:
```

```
        print("Quizzes")
    case 4:
        print("Blog")
    case _:
        print("Invalid option")
```

Output:

```
Resources
```

This is useful for menu systems.

---

12. User Input Example

You can combine `match...case` with `input()`.

```
choice = input("Choose an option: ")

match choice:
    case "1":
        print("Opening Home")
    case "2":
        print("Opening Resources")
    case "3":
        print("Opening Quizzes")
    case "4":
        print("Opening Blog")
    case _:
        print("Invalid choice")
```

Notice that the cases are strings:

```
case "1":
```

because `input()` returns a string.

---

13. Converting User Input to an Integer

Alternatively:

```
choice = int(input("Choose an option: "))

match choice:
    case 1:
        print("Opening Home")
```

```
    case 2:
        print("Opening Resources")
    case 3:
        print("Opening Quizzes")
    case 4:
        print("Opening Blog")
    case _:
        print("Invalid choice")
```

Now the cases are integers.

---

14. haas.dev Example: Resource Actions

Imagine a resource page where a user can perform actions.

```
action = "download"

match action:
    case "view":
        print("Open resource")
    case "download":
        print("Download PDF")
    case "share":
        print("Share resource")
    case "bookmark":
        print("Bookmark resource")
    case _:
        print("Unknown action")
```

This maps the possible application actions directly to code.

---

15. haas.dev Example: Resource Status

Suppose a resource can have several statuses:

```
status = "published"

match status:
    case "draft":
        print("Resource is still being prepared")
    case "published":
        print("Resource is live")
    case "archived":
```

```
        print("Resource is archived")
    case _:
        print("Unknown resource status")
```

This is a clean way to represent a fixed set of states.

---

16. Matching Multiple Values

One powerful feature is that a single case can match multiple alternatives.

For example:

```
day = "Saturday"

match day:
    case "Saturday" | "Sunday":
        print("Weekend")
    case _:
        print("Weekday")
```

The `|` means:

> **Match either this pattern OR that pattern.**

So both:

```
"Saturday"
```

and:

```
"Sunday"
```

match the same case.

---

17. Multiple Alternatives

You can have more alternatives:

```
command = "quit"

match command:
    case "quit" | "exit" | "q":
        print("Exiting program")
    case "help" | "h":
        print("Showing help")
    case _:
        print("Unknown command")
```

This can be cleaner than writing:

```
if command == "quit" or command == "exit" or command == "q":
```

---

18. Matching Boolean Values

You can also match Boolean values.

```
is_logged_in = True

match is_logged_in:
    case True:
        print("Show dashboard")
    case False:
        print("Show login")
```

However, for a simple Boolean decision, regular `if` is usually clearer:

```
if is_logged_in:
    print("Show dashboard")
else:
    print("Show login")
```

This is an important lesson:

> **Just because a feature can do something doesn't mean it is the best tool for every situation.**

---

19. Matching `None`

`match` can also distinguish `None`.

```
user = None

match user:
    case None:
        print("No user found")
    case _:
        print("User exists")
```

This can be useful when handling optional values.

---

20. Matching Lists

This is where `match...case` becomes much more powerful.

You can match the structure of a list.

```
numbers = [1, 2]

match numbers:
    case []:
        print("Empty list")
    case [x]:
        print("One item")
    case [x, y]:
        print("Two items")
    case _:
        print("More items")
```

For:

```
numbers = [1, 2]
```

the matching case is:

```
case [x, y]:
```

---

21. Capturing Values

In:

```
case [x, y]:
```

`x` and `y` capture the values from the list.

For:

```
numbers = [10, 20]
```

Python effectively matches:

```
x = 10
y = 20
```

Then:

```
case [x, y]:
    print(x)
    print(y)
```

produces:

```
10
20
```

This is one of the features that makes pattern matching different from a simple collection of `if` statements.

---

22. Matching the Shape of Data

Consider:

```
data = ["Hafsa", 25]
```

You can write:

```
match data:
    case [name, age]:
        print(f"Name: {name}")
        print(f"Age: {age}")
```

Python doesn't only ask:

> "Is the value equal to something?"

It can ask:

> **"Does the value have this structure?"**

This is called **structural pattern matching**.

---

23. Matching Lists With Specific Values

You can combine values and variables.

```
command = ["move", "left"]

match command:
    case ["move", "left"]:
        print("Moving left")
    case ["move", "right"]:
        print("Moving right")
    case ["move", direction]:
        print(f"Moving {direction}")
    case _:
        print("Unknown command")
```

The pattern itself describes the expected structure.

---

24. Matching a List With a Rest Pattern

You can use `*` to capture multiple remaining items.

```
numbers = [1, 2, 3, 4]

match numbers:
    case [first, *rest]:
        print("First:", first)
        print("Rest:", rest)
```

Output:

```
First: 1
Rest: [2, 3, 4]
```

This is useful when the exact number of remaining elements isn't known.

---

25. Matching Tuples

Tuples can also be matched.

```
point = (10, 20)

match point:
    case (0, 0):
        print("Origin")
    case (x, 0):
        print("On X-axis")
    case (0, y):
        print("On Y-axis")
    case (x, y):
        print("Somewhere else")
```

For:

```
point = (10, 20)
```

the final case matches.

---

26. Matching Dictionaries

`match...case` can also work with dictionaries.

```
user = {
    "name": "Hafsa",
    "role": "student"
}

match user:
    case {"role": "admin"}:
        print("Admin user")
    case {"role": "student"}:
        print("Student user")
    case _:
        print("Unknown role")
```

This is useful when working with structured data.

---

27. Dictionary Patterns

You can capture values from dictionaries.

```
user = {
    "name": "Hafsa",
    "role": "student"
}

match user:
    case {"name": name, "role": role}:
        print(name)
        print(role)
```

The variables capture values from matching keys.

---

28. A Powerful Concept: Shape + Value

Pattern matching can consider both:

Value

```
case "login":
```

and:

Structure

```
case [name, age]:
```

and:

Structure + specific values

```
case {"role": "admin"}:
```

This is why Python calls the feature **structural pattern matching**.

---

29. Guards With `if`

`case` can have an additional condition called a **guard**.

Syntax:

```
match value:
    case pattern if condition:
        # code
```

Example:

```
age = 25

match age:
    case age if age < 18:
        print("Minor")
    case age if age < 60:
        print("Adult")
    case _:
        print("Senior")
```

The pattern matches first, and the guard provides an additional condition.

---

30. A Simpler Guard Example

```
number = 10

match number:
    case x if x > 0:
        print("Positive")
    case x if x < 0:
        print("Negative")
    case 0:
        print("Zero")
```

Output:

```
Positive
```

Here:

```
case x if x > 0:
```

means:

> Match the value as `x`, but only use this case if `x > 0`.

---

31. `match...case` Is Not Just Another `switch`

Many programming languages have a `switch` statement.

Python's `match...case` can look similar:

```
match command:
    case "start":
        print("Starting")
    case "stop":
        print("Stopping")
```

But Python's feature is more powerful because it supports **patterns**, not just fixed values.

It can match:

- literals
- sequences
- mappings
- classes
- alternatives
- captured variables
- guarded patterns

So it is better understood as **pattern matching**, not simply a replacement for `switch`.

---

32. `match...case` vs `if...elif...else`

Use `if...elif...else` when the logic is primarily about conditions.

Example:

```
if age >= 18:
    print("Adult")
elif age >= 13:
    print("Teenager")
```

```
else:
    print("Child")
```

Use `match...case` when you are primarily matching a value or structure.

Example:

```
match role:
    case "admin":
        print("Admin")
    case "student":
        print("Student")
    case "guest":
        print("Guest")
```

This distinction will help you choose the appropriate tool.

---

33. Fixed Values Are a Good Match for `match`

Suppose you have:

```
pending
approved
rejected
cancelled
```

A `match` structure communicates this clearly:

```
match status:
    case "pending":
        print("Waiting")
    case "approved":
        print("Approved")
    case "rejected":
        print("Rejected")
    case "cancelled":
        print("Cancelled")
```

The code reads almost like the list of possible states.

---

34. Complex Numeric Conditions Usually Fit `if`

For example:

```
if score >= 90:
    print("A")
elif score >= 80:
    print("B")
elif score >= 70:
    print("C")
```

This is naturally expressed with comparisons.

You could use guards with `match`, but it may not make the code better.

```
match score:
    case x if x >= 90:
        print("A")
    case x if x >= 80:
        print("B")
    case x if x >= 70:
        print("C")
```

This works, but `if...elif` is arguably more direct for this kind of range logic.

The lesson:

> **Choose syntax based on the shape of the problem, not because a feature is newer or more advanced.**

---

35. Real-World Example: API Response Status

Imagine an API returns a status code.

```
status_code = 404

match status_code:
    case 200:
        print("Request successful")
    case 201:
        print("Resource created")
    case 400:
        print("Bad request")
    case 401:
        print("Authentication required")
    case 403:
        print("Access denied")
    case 404:
        print("Resource not found")
```

```
    case _:
        print("Other response")
```

This is a natural use case because the program is matching known values.

---

36. Real-World Example: Payment Status

```
payment_status = "failed"

match payment_status:
    case "pending":
        print("Payment is processing")
    case "completed":
        print("Payment successful")
    case "failed":
        print("Payment failed")
    case "refunded":
        print("Payment refunded")
    case _:
        print("Unknown payment status")
```

The code directly communicates the possible states.

---

37. Real-World Example: Mobile App Navigation

Suppose an application receives navigation commands.

```
screen = "settings"

match screen:
    case "home":
        print("Show home screen")
    case "profile":
        print("Show profile screen")
    case "resources":
        print("Show resources screen")
    case "settings":
        print("Show settings screen")
    case _:
        print("Screen not found")
```

This is useful for applications that need to react to predefined states or commands.

---

38. Real-World Example: haas.dev Content Types

Imagine haas.dev has different content types:

```
content_type = "pdf"

match content_type:
    case "pdf":
        print("Open learning PDF")
    case "quiz":
        print("Open quiz")
    case "blog":
        print("Open blog article")
    case "tool":
        print("Open developer tool")
    case _:
        print("Unknown content type")
```

Each content type has a different behavior.

---

39. Matching Multiple Resource Types

You can combine related cases:

```
content_type = "pdf"

match content_type:
    case "pdf" | "ebook" | "guide":
        print("Open document")
    case "quiz" | "assessment":
        print("Open quiz")
    case "blog" | "article":
        print("Open article")
    case _:
        print("Unknown content")
```

This avoids repeating the same action.

---

40. Real-World Example: Command Processor

Imagine a command-line tool.

```
command = "help"
```

```
match command:
    case "start":
        print("Starting application")
    case "stop":
        print("Stopping application")
    case "restart":
        print("Restarting application")
    case "help" | "h":
        print("Showing help")
    case "quit" | "exit":
        print("Exiting")
    case _:
        print("Unknown command")
```

This is a strong use case for `match...case`.

---

41. Nested `match`

You can nest `match` statements just like you can nest `if` statements.

```
user_type = "student"
action = "download"

match user_type:
    case "admin":
        print("Admin access")
    case "student":
        match action:
            case "view":
                print("View resource")
            case "download":
                print("Download resource")
            case _:
                print("Unknown action")
    case _:
        print("Unknown user")
```

However, the same principle applies:

> Don't create unnecessary nesting.

If the logic becomes difficult to follow, simplify it.

---

42. Combining `match` With Conditions

You can use guards to represent more detailed rules.

```
user = {
    "role": "student",
    "age": 20
}

match user:
    case {"role": "student", "age": age} if age >= 18:
        print("Adult student")
    case {"role": "student"}:
        print("Student")
    case {"role": "admin"}:
        print("Admin")
    case _:
        print("Unknown user")
```

This demonstrates how pattern structure and additional conditions can work together.

---

43. Common Mistake: Forgetting `case`

Incorrect:

```
match command:
    "login":
        print("Login")
```

Correct:

```
match command:
    case "login":
        print("Login")
```

Every pattern must be introduced with:

```
case
```

---

44. Common Mistake: Forgetting the Colon

Incorrect:

```
match command
```

Correct:

```
match command:
```

Likewise:

```
case "login":
```

---

45. Common Mistake: Wrong Indentation

Incorrect:

```
match command:
case "login":
    print("Login")
```

Correct:

```
match command:
    case "login":
        print("Login")
```

The `case` blocks belong inside the `match`.

---

46. Common Mistake: Using `match` on an Older Python Version

`match...case` requires:

```
Python 3.10+
```

If you are using an older Python version, this syntax will not work.

Check your version from the terminal:

```
python --version
```

or:

```
python3 --version
```

If your environment uses Python 3.10 or newer, `match...case` is available.

---

47. Common Mistake: Expecting `case` to Work Like Normal Equality Everywhere

Pattern matching has its own rules.

For straightforward fixed values:

```
  case "login":
```

is easy to understand.

But once you start using variables, sequences, mappings, and class patterns, `case` is doing pattern matching rather than simply acting like:

```
  value == something
```

This distinction becomes important as you use advanced patterns.

---

48. Common Mistake: Putting a Wildcard Too Early

Consider:

```
match command:
    case _:
        print("Unknown")
    case "login":
        print("Login")
```

The wildcard matches everything.

Therefore, the `"login"` case can never be reached.

Correct:

```
match command:
    case "login":
        print("Login")
    case "register":
        print("Register")
    case _:
        print("Unknown")
```

General rule:

> **Put broad patterns after specific patterns.**

---

49. Common Mistake: Using `match` Everywhere

You don't need to replace every `if` statement with `match`.

This:

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

is perfectly clear.

Using:

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

doesn't necessarily improve it.

Use the tool that makes the logic easiest to understand.

---

50. Common Mistake: Forgetting the Default Case

This is valid:

```
match command:
    case "login":
        print("Login")
    case "logout":
        print("Logout")
```

But if an unexpected command appears, nothing happens.

Often you want:

```
case _:
    print("Unknown command")
```

Whether you need a default case depends on the program.

---

51. Pattern Matching as Data Modeling

One of the biggest ideas behind `match...case` is that your program can respond to the **shape of data**.

For example:

```
event = ["login", "Hafsa"]
```

You could write:

```
match event:
    case ["login", username]:
        print(f"{username} logged in")
    case ["logout", username]:
        print(f"{username} logged out")
    case _:
        print("Unknown event")
```

The program is matching both:

```
event type
```

and:

```
event structure
```

This becomes useful when working with structured application data.

---

52. Mini Quiz

Question 1

What does this do?

```
match command:
    case "start":
        print("Starting")
    case "stop":
        print("Stopping")
    case _:
        print("Unknown")
```

It checks the value of `command` against the patterns and uses the first matching case.

Question 2

What does this mean?

```
case "yes" | "y":
```

It matches either `"yes"` or `"y"`.

Question 3

What does this mean?

```
case _:
```

It is a wildcard pattern that can match anything not matched by an earlier case.

---

53. Five-Minute Challenge

Build a **Simple Command Processor**.

The user should enter:

```
start
stop
restart
help
exit
```

Use `match...case`.

Expected behavior:

```
start   → Starting application
stop    → Stopping application
restart → Restarting application
help    → Showing help
exit    → Exiting application
other   → Unknown command
```

Bonus:

Allow:

```
h
q
```

as shortcuts for:

```
help
exit
```

Use:

```
|
```

to match alternatives.

---

54. Practice Project: haas.dev Resource Router

Build a simple resource router.

Ask the user to enter a content type:

```
pdf
quiz
blog
tool
```

Then use `match...case`:

```
content_type = input("Enter content type: ")

match content_type:
    case "pdf":
        print("Opening PDF resource")
    case "quiz":
        print("Opening quiz")
    case "blog":
        print("Opening blog")
    case "tool":
        print("Opening developer tool")
    case _:
        print("Unknown content type")
```

Then extend it.

Allow:

```
guide
ebook
```

to behave like:

```
pdf
```

using:

```
case "pdf" | "guide" | "ebook":
```

---

55. Practice Project: API Response Handler

Create a program that accepts a status code.

```
status_code = int(input("Enter status code: "))
```

Handle:

```
200 → Success
201 → Created
400 → Bad Request
401 → Unauthorized
403 → Forbidden
404 → Not Found
500 → Server Error
```

Use `match...case`.

Add:

```
case _:
```

for unknown status codes.

---

56. Practice Project: Menu System

Build:

```
1 → Home
2 → Resources
3 → Quizzes
4 → Blog
5 → Settings
```

Use:

```
choice = int(input("Choose an option: "))
```

Then:

```
match choice:
    case 1:
        print("Home")
    case 2:
        print("Resources")
    case 3:
        print("Quizzes")
    case 4:
        print("Blog")
    case 5:
        print("Settings")
    case _:
        print("Invalid option")
```

This is one of the simplest ways to practice `match...case`.

---

57. Debugging `match...case`

When a case isn't behaving as expected, check:

1. What is the actual value?

```
print(command)
```

2. What is its type?

```
print(type(command))
```

This matters because:

```
"1"
```

and:

```
1
```

are different values.

For example:

```
choice = input("Choice: ")

match choice:
    case 1:
        print("One")
```

will not match when the user enters `"1"` because `input()` produces a string.

Use:

```
case "1":
```

or convert the input:

```
choice = int(input("Choice: "))
```

and then use:

```
case 1:
```

---

58. Testing `match...case`

If you have:

```
match status:
    case "pending":
        print("Pending")
    case "approved":
        print("Approved")
    case "rejected":
        print("Rejected")
    case _:
        print("Unknown")
```

test at least:

```
pending
approved
rejected
something unexpected
```

Testing only the expected values isn't enough.

The wildcard/default case should also be tested.

---

59. `match...case` and Clean Code

Good code should make the possible states obvious.

Compare:

```
if status == "pending":
    ...
elif status == "approved":
    ...
elif status == "rejected":
    ...
elif status == "cancelled":
    ...
```

with:

```
match status:
    case "pending":
        ...
    case "approved":
        ...
    case "rejected":
        ...
```

```
    case "cancelled":
        ...
```

When you are dealing with a known set of states, the second structure can communicate the model very clearly.

But the first is still perfectly valid.

There is no rule saying one must always replace the other.

---

60. Engineering Thinking: Identify the Shape of the Problem

Before choosing `if` or `match`, ask:

Is this primarily a condition?

For example:

```
If score is above 90...
If age is below 18...
If balance is greater than price...
```

Use `if`.

Is this primarily a known value or pattern?

For example:

```
If status is "pending"...
If command is "login"...
If event looks like ["login", username]...
```

`match...case` may be a natural fit.

This distinction will help you write clearer Python.

---

61. A More Advanced Example

Suppose an application receives events:

```
event = ["login", "Hafsa"]
```

You can process different event structures:

```
match event:
    case ["login", username]:
        print(f"{username} logged in")

    case ["logout", username]:
```

```
        print(f"{username} logged out")

    case ["download", resource]:
        print(f"Downloaded {resource}")

    case ["error", code]:
        print(f"Error: {code}")

    case _:
        print("Unknown event")
```

This is much more than checking one value.

The program is recognizing the structure of the incoming data.

---

62. Why Pattern Matching Matters

At beginner level, you may see:

```
match value:
    case 1:
    case 2:
    case 3:
```

and think:

> "This is just another way to write `if`."

For simple values, that is a reasonable observation.

But the deeper capability is:

```
Match the shape of data
        +
Match specific values
        +
Capture useful values
        +
Apply additional conditions
```

That is why `match...case` becomes useful in more advanced Python programs.

---

63. Key Takeaways

After this lesson, you should understand:

- `match...case` was introduced in Python 3.10.

- It implements structural pattern matching.
- `match` evaluates an expression.
- `case` defines patterns to match.
- Python checks cases in order.
- The first matching case is selected.
- `case _:` acts as a wildcard/default case.
- `|` allows multiple alternatives in one case.
- Patterns can match strings, numbers, lists, tuples, dictionaries, and more.
- Variables can capture values from matched structures.
- Guards allow additional conditions with `if`.
- `match...case` can represent application states and commands clearly.
- `if...elif...else` remains better for many ordinary conditional/range problems.
- The goal is not to use `match` everywhere.
- The goal is to choose the structure that best represents the problem.

---

64. Cheat Sheet

Basic matching

```
match value:
    case "a":
        print("A")
    case "b":
        print("B")
    case _:
        print("Other")
```

Multiple alternatives

```
match command:
    case "quit" | "exit" | "q":
        print("Exit")
```

List pattern

```
match data:
    case [x, y]:
        print(x, y)
```

Capture remaining items

```
match data:
    case [first, *rest]:
        print(first)
        print(rest)
```

Dictionary pattern

```
match user:
    case {"role": "admin"}:
        print("Admin")
```

Guard

```
match number:
    case x if x > 0:
        print("Positive")
    case x if x < 0:
        print("Negative")
    case 0:
        print("Zero")
```

Default case

```
case _:
    print("Unknown")
```

---

65. Final Practice Task

Build a **haas.dev Command and Resource Router**.

Your program should accept commands such as:

```
pdf
quiz
blog
tool
help
exit
```

Use `match...case` to decide what happens.

Then make it more advanced.

Support:

```
pdf
guide
ebook
```

as document resources.

Support:

```
quiz
assessment
```

as quiz resources.

Support:

```
blog
article
```

as article resources.

Then add structured input such as:

```
event = ["download", "python"]
```

and use pattern matching to detect:

```
download → resource name
view → resource name
share → resource name
```

The goal is to move from simply matching values to understanding how Python can **recognize patterns in data and choose behavior based on those patterns**.

---

3 Questions Before You Touch the Keyboard

1. Am I matching a known value, or am I evaluating a condition?
2. Could this problem be represented more clearly as patterns or states?
3. What should happen when no expected pattern matches?

Visit haas.dev for more resources and guides.

haas.dev

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