

Day 4 Making decisions

Self-Study Guide · About 60 minutes · With answer key

By the end you will be able to

- Write comparisons and combine them with and, or, and not, including chained comparisons such as $0 < x < 10$.
- Translate a set of rules written in English into an if, elif, and else chain.
- Predict which values count as true or false in a condition, and check for None with is.
- Pick between two values with a conditional expression, and match one value against fixed choices with match and case.

What to remember from earlier

- Indentation is part of Python's syntax, and today you use it to group the lines that belong together. ([Day 1: Setup and your first program, 1.7 Indentation Is Part of the Syntax](#))
- A bool is either True or False, and None means "no value here". ([Day 2: Variables, numbers and basic types, 2.6 True, False, and None](#))
- The remainder operator % gives what is left over, so an even number has remainder 0 when divided by 2. ([Day 2: Variables, numbers and basic types, 2.4 Floor Division, Remainders, and Powers](#))
- == checks whether two strings are exactly the same, so capital letters count. ([Day 3: Strings, 3.2 Length, Concatenation, and in](#))
- strip() removes spaces from the ends of typed text, and lower() turns it into small letters. ([Day 3: Strings, 3.6 String Methods](#))
- An f-string, like f"Hi, {name}!", puts values into text. ([Day 3: Strings, 3.8 f-strings and Format Specifiers](#))
- input() always gives back a string, even when the user types digits. ([Day 3: Strings, 3.9 Reading Input](#))

Make a `day04` folder for today's files, open it in your editor, and run each script from its terminal, as on Days 1 to 3.

Every program so far has run every line, top to bottom, every time. Real programs make choices: print a different grade for a different score, charge a child less than an adult, answer the command the user typed. Today you learn to ask questions about values and to run code only when the answer is yes.

4.1 Comparison Operators

To decide anything, a program first needs a question with a yes-or-no answer: is the score at least 50? Is the password the one we expected? On Day 2 you met the two values that answer such questions, `True` and `False`, and on Day 3 you used `==` to check whether two strings match. Python has six operators that ask questions like these.

A **comparison operator** compares two values and gives a bool: `True` if the comparison holds, `False` if it does not. An expression that gives a bool is called a **condition**.

Operator	Asks	Example	Result
<code>==</code>	Equal to?	<code>5 == 5</code>	True
<code>!=</code>	Not equal to?	<code>5 != 5</code>	False
<code><</code>	Less than?	<code>3 < 8</code>	True
<code>></code>	Greater than?	<code>3 > 8</code>	False
<code><=</code>	Less than or equal to?	<code>8 <= 8</code>	True
<code>>=</code>	Greater than or equal to?	<code>7 >= 8</code>	False

Two-character operators are typed with no space between the characters, and the `=` always comes second: `<=`, not `<.` Read `!=` as "is not equal to"; the `!` means "not".

```
>>> score = 72
>>> score > 50
True
>>> score < 50
False
>>> score == 72
True
>>> score != 72
False
>>> score >= 72
True
>>> score <= 71
False
```

`score` is 72, so `score > 50` is True. `score >= 72` is True because "or equal to" counts 72 itself, while `score <= 71` is False.

A comparison is an expression like any other, so you can store its result in a variable, print it, or compare the result of a calculation:

WORKED EXAMPLE Weather flags

`weather_flags.py`

```
1 temperature = 31.5
2 is_hot = temperature > 30
3 is_freezing = temperature <= 0
4 print("Hot:", is_hot)
5 print("Freezing:", is_freezing)
6 print("Feels like summer:", temperature + 2 >= 30)
```

```
Hot: True
Freezing: False
Feels like summer: True
```

Line 2 works out `31.5 > 30`, which is True, and stores that bool in `is_hot`. Line 3 stores False. In line 6, Python does the arithmetic **before** the comparison: `temperature + 2` is 33.5, then `33.5 >= 30` is True. Comparisons have lower precedence than every arithmetic operator from Day 2, so `a + b > c` always means `(a + b) > c`.

Comparisons also work across types and on strings:

```
>>> 7 == 7.0
True
>>> 5 == "5"
False
>>> "apple" < "banana"
True
>>> "cat" < "catch"
True
>>> "Zebra" < "apple"
True
```

- An int and a float compare by value, so `7 == 7.0` is `True`.
- A number never equals a string, so `5 == "5"` is `False`. No error, just `False`.
- Strings compare **alphabetically**, character by character from the left. `"apple" < "banana"` because `a` comes before `b`. When one string runs out first, as `"cat"` does against `"catch"`, the shorter one comes first.
- Every capital letter comes before every small letter, so `"Zebra" < "apple"` is `True`. Python compares the code number that the computer stores for each character, and the capitals have the smaller numbers. Use `lower()` on both sides when you want a true dictionary order.

KEY IDEA

`==`, `!=`, `<`, `>`, `<=`, and `>=` compare two values and give `True` or `False`. Arithmetic happens first. Strings compare alphabetically, with capitals before small letters.

WATCH OUT

Comparing typed input with a number. `input()` returns a string (Day 3), and Python refuses to say whether a string is greater than a number:

age_check.py

```
1 age = input("Age: ")
2 print("Adult:", age >= 18)
```

```
Age: 15
Traceback (most recent call last):
  File "../day04/age_check.py", line 2, in <module>
    print("Adult:", age >= 18)
                        ^^^^^^^^^
TypeError: '>=' not supported between instances of 'str' and 'int'
```

`age` is `"15"`, a `str`, and `18` is an `int`. Unlike `==`, which just says `False`, the ordering operators stop with a `TypeError`. Convert as you read: `age = int(input("Age: "))`.

WATCH OUT

Testing floats for exact equality. Day 2 showed that `0.1 + 0.2` is stored as a value a tiny bit off from `0.3`, so `==` says they differ:

```
>>> 0.1 + 0.2 == 0.3
False
>>> round(0.1 + 0.2, 2) == 0.3
True
```

Round both sides to the number of places you care about before comparing them, or use `<` and `>` instead of `==` with floats.

PAUSE AND TRY**Question 1** Tracing

Predict exactly what the REPL prints for each line.

```
>>> price = 4.5
>>> price >= 4.5
>>> price * 2 != 9
>>> "Dog" == "dog"
>>> "ant" < "Bee"
>>> 10 == 10.0
```

4.2 Chained Comparisons

A water park ride is for ages 13 to 19. With one comparison you can check "at least 13" or "at most 19", but not both. In maths you would write $13 \leq \text{age} \leq 19$, and Python lets you write exactly that.

A **chained comparison** puts several comparisons in a row, such as `13 <= age <= 19`. It is `True` only when **every** link in the chain is true: `13 <= age` **and** `age <= 19`. The value in the middle is used in both links.

```
>>> age = 15
>>> 13 <= age <= 19
True
>>> age = 25
>>> 13 <= age <= 19
False
>>> 0 < 5 < 10 < 20
True
```

With `age = 15`, both `13 <= 15` and `15 <= 19` hold, so the chain is `True`. With 25 the second link, `25 <= 19`, fails, so the whole chain is `False`. A chain can be any length: `0 < 5 < 10 < 20` checks three links.

Read a chain like a number line, smallest on the left: `low <= value <= high` means "value is between low and high". Use `<` at an end to leave that end out: `0 < minutes < 60` allows 1 to 59 when `minutes` is a whole number.

KEY IDEA

`a < b < c` means `a < b` and `b < c`. Write a range check as `low <= value <= high`, with the value in the middle.

WATCH OUT

Leaving the value out of the middle. "Age is at least 13 and at most 19" is sometimes typed as `age >= 13 <= 19`:

```
>>> age = 30
>>> age >= 13 <= 19
True
>>> 13 <= age <= 19
False
```

The chain's links are `age >= 13` and `13 <= 19`. The second link compares 13 with 19 and is always true, so any age of 13 or more passes, even 30. Put the value being tested in the middle: `13 <= age <= 19`.

PAUSE AND TRY**Question 2** Multiple choice

`minutes` holds a whole number. Which condition is `True` exactly when `minutes` is from 1 to 59?

- A. `1 < minutes < 59`
- B. `0 < minutes < 60`
- C. `minutes > 0 < 60`
- D. `60 > minutes > 1`

4.3 Combining Conditions: and, or, not

Many rules need more than one condition. "You can ride if you are tall enough **and** old enough." "The shop is closed on Saturday **or** Sunday." A chain handles ranges of one value, but these rules join conditions about different things. Python joins conditions with three words.

A **logical operator** combines bools into a new bool:

- **and** gives `True` only when **both** sides are `True`.
- **or** gives `True` when **at least one** side is `True`. (It is also `True` when both are; in everyday English "or" sometimes means "one but not both", but Python's `or` never does.)
- **not** goes in front of a single bool and flips it: `not True` is `False`.

A **truth table** lists every combination:

a	b	a and b	a or b
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

```
>>> age = 16
>>> has_ticket = True
>>> age >= 18 and has_ticket
False
>>> age >= 18 or has_ticket
True
>>> not has_ticket
False
>>> not age >= 18
True
```

`age >= 18` is `False`, so `and` gives `False` even though `has_ticket` is `True`, while `or` needs only one, so it gives `True`. In `not age >= 18`, the comparison happens first, giving `False`, and `not` flips it to `True`.

The chained comparison from 4.2 is a short way of writing `and`: `13 <= age <= 19` means the same as `13 <= age and age <= 19`.

WORKED EXAMPLE Can this child ride?

A ride requires a height of at least 120 cm, and the rider must be 10 or older **or** come with an adult.

can_ride.py

```
1 height = 125
2 age = 9
3 with_adult = True
4 tall_enough = height >= 120
5 can_ride = tall_enough and (age >= 10 or with_adult)
6 print("Tall enough:", tall_enough)
7 print("Can ride:", can_ride)
```

```
Tall enough: True
Can ride: True
```

Step	Worked out	Value
1	<code>125 >= 120</code>	<code>True</code>
2	<code>9 >= 10</code>	<code>False</code>
3	<code>False or True</code> (in parentheses)	<code>True</code>
4	<code>True and True</code>	<code>True</code>

The parentheses in line 5 make Python work out the `or` first, exactly as parentheses do in arithmetic.

Without parentheses, Python groups the operators in a fixed order: comparisons bind tightest, then `not`, then `and`, then `or`. Think of `and` as multiplication and `or` as addition: `and` binds tighter, so `a or b and c` means `a or (b and c)`. Grouping only decides which parts belong together; Python still works out the parts from left to right, as 4.4 shows.

KEY IDEA

`and` needs both sides true, `or` needs at least one, and `not` flips a bool. Comparisons bind tightest, then `not`, then `and`, then `or`; use parentheses to make the grouping you mean certain.

WATCH OUT

Mixing `and` and `or` without parentheses. The rule is "go to the park if it is the weekend or a holiday, and it is not raining". It is the weekend, and it is raining:

```
>>> is_weekend = True
>>> is_holiday = False
>>> is_raining = True
>>> is_weekend or is_holiday and not is_raining
True
>>> (is_weekend or is_holiday) and not is_raining
False
```

The first line says `True`: go to the park in the rain. Because `and` binds tighter, Python reads it as `is_weekend or (is_holiday and not is_raining)`, and `is_weekend` alone makes it true. The second line puts parentheses around the `or` part, so the rain check applies to both, and gives the correct `False`. Whenever a condition mixes `and` with `or`, add parentheses.

PAUSE AND TRY**Question 3** Tracing

Predict exactly what the REPL prints for each line.

```
>>> x = 7
>>> x > 5 and x < 10
>>> x < 5 or x > 10
>>> not x == 7
>>> x > 5 or x > 100 and x < 0
```

4.4 Short-Circuit Evaluation

Suppose you want to know whether a class's average score is over 50, but the class might have no scores yet. Working out the average means dividing by the count, and Day 2 showed that dividing by zero stops the program. You need the check "is there anything to average?" to protect the division.

Look at the truth table in 4.3 again. If the left side of `and` is `False`, the answer is `False` whatever the right side is. If the left side of `or` is `True`, the answer is `True` whatever the right side is. Python takes advantage of this: it works out the left side first, and when that already decides the answer, it **does not run the right side at all**. This is called **short-circuit evaluation**.

`short_circuit.py`

```
1 total = 0
2 count = 0
3 print(count > 0 and total / count > 50)
4 print(count == 0 or total / count > 50)
```

False
True

In line 3, `count > 0` is `False`, so `and` stops there and gives `False`; the division `total / count` never runs. In line 4, `count == 0` is `True`, so `or` stops and gives `True`. Neither line divides by zero.

Now swap the order, so the division comes first:

`short_circuit_crash.py`

```
1 total = 0
2 count = 0
3 print(total / count > 50 and count > 0)
```

```
Traceback (most recent call last):
  File ".../day04/short_circuit_crash.py", line 3, in <module>
    print(total / count > 50 and count > 0)
          ~~~~~^~~~~~
ZeroDivisionError: division by zero
```

The left side runs first, and it divides by zero before Python ever reaches `count > 0`. The same two conditions are safe in one order and a crash in the other.

KEY IDEA

Python works out `and` and `or` left to right and stops as soon as the answer is known: `and` stops at a `False` (or any falsy value, from 4.7), and `or` stops at a `True` (or any truthy value). Put the safety check first, as in `count > 0 and total / count > 50`.

PAUSE AND TRY

Question 4 Short answer

`count` is the number of tests taken and `total` the sum of their scores. Explain why `count != 0 and total / count >= 70` is safe when `count` is 0, but `total / count >= 70 and count != 0` is not. Do both give the same answer when `count` is 4?

4.5 if Statements and Blocks

You can now ask questions, but every line still runs whatever the answer. To print "Drink plenty of water" only on hot days, a program needs to **skip** lines when a condition is `False`. Day 1 said indentation shows which lines belong together in a group, and that you would build those groups on Day 4. Here they are.

An **if statement** runs a group of lines only when its condition is `True`. Its shape:

```
1 if condition:
2     statement
3     statement
```

- `if` is a keyword (Day 2 listed it), followed by the condition.
- The **colon** `:` ends the line and means "a group of lines follows".
- The indented lines below form a **block**: the lines that belong to the `if`. Python style uses 4 spaces per level; most editors indent the next line for you when you press Enter after a colon.
- The block ends at the first line that goes back to the left edge. That line is not part of the `if`, so it always runs.

Choosing between paths like this is called **selection**, and a block can hold as many lines as you need.

WORKED EXAMPLE A hot day

hot_day.py

```
1 temperature = 31
2 print("Checking the weather...")
3 if temperature > 30:
4     print("It is hot today.")
5     print("Drink plenty of water.")
6 print("Have a good day!")
```

```
Checking the weather...
It is hot today.
Drink plenty of water.
Have a good day!
```

Line 2 always runs. Line 3 asks `31 > 30`, which is `True`, so Python runs the block, lines 4 and 5. Line 6 is back at the left edge, so it runs after the `if` is finished either way.

Now change only the temperature, to 18:

cool_day.py

```
1 temperature = 18
2 print("Checking the weather...")
3 if temperature > 30:
4     print("It is hot today.")
5     print("Drink plenty of water.")
6 print("Have a good day!")
```

```
Checking the weather...
Have a good day!
```

`18 > 30` is `False`, so Python skips the whole block, both indented lines, and continues at line 6. The indentation is what tells Python that line 5 belongs to the `if` and line 6 does not.

If you forget to indent the block, Python cannot tell that the line belongs to the `if`, and nothing runs:

`if_no_indent.py`

```
1 score = 75
2 if score >= 50:
3 print("Pass")
```

```
File ".../day04/if_no_indent.py", line 3
    print("Pass")
    ^^^^^
```

IndentationError: expected an indented block after 'if' statement on line 2

KEY IDEA

`if condition:` runs the indented block below it only when the condition is `True`. The block ends at the first line back at the left edge, and that line always runs.

WATCH OUT

Writing = in a condition. `=` assigns a value (Day 2); `==` compares. In a condition, a single `=` is a syntax error:

`if_assign.py`

```
1 score = 75
2 if score = 100:
3     print("Perfect score!")
```

```
File ".../day04/if_assign.py", line 2
    if score = 100:
        ^^^^^^^^^^^
```

SyntaxError: invalid syntax. Maybe you meant '==' or ':=' instead of '='?

Python even suggests the fix. (Ignore `:=`; it is something else and not used in this course.) Write `if score == 100:`. Older versions of Python may word this message differently.

PAUSE AND TRY

Question 5 Tracing

What does this program print? Watch the value of `money` change inside the first block.

shoes.py

```
1 money = 20
2 price = 15
3 print("Looking at the shoes...")
4 if money >= price:
5     print("Buying them.")
6     money = money - price
7 if money < price:
8     print("Saving up instead.")
9 print("Money left:", money)
```

Question 6 Debugging

This program should print a library fine of 0.25 per day late. It stops before printing anything. Quote the last line of the error, explain it, and write the fix.

library_fine.py

```
1 days_late = 4
2 if days_late > 0
3     print("Fine:", days_late * 0.25)
4 print("Thank you!")
```

4.6 else and elif

Question 5 showed a problem: two separate `if` statements are two separate questions, and both can run. Often you want **exactly one** of two paths: buy the shoes **or** save up, never both.

An **else** clause gives an `if` a second block that runs when the condition is `False`. `else` has no condition of its own, sits at the same indentation as its `if`, and ends with a colon:

shoes_else.py

```
1 money = 20
2 price = 15
3 print("Looking at the shoes...")
4 if money >= price:
5     print("Buying them.")
6     money = money - price
7 else:
8     print("Saving up instead.")
9 print("Money left:", money)
```

```
Looking at the shoes...
Buying them.
Money left: 5
```

`20 >= 15` is `True`, so the `if` block runs and the `else` block is skipped. The condition is checked once, before `money` changes, so the purchase cannot trigger the second message. With `money = 10`, the `if` block would be skipped and the `else` block would run instead. One of the two always runs, never both.

For more than two paths, add **elif** ("else if") clauses between the **if** and the **else**. Each **elif** has its own condition. Together they form an **if chain**: Python checks the conditions **from the top down**, runs the block of the **first** one that is **True**, and skips everything else in the chain. The **else** at the bottom catches every case that no condition matched. It is optional; without it, a chain where nothing matches runs no block at all.

WORKED EXAMPLE What to wear

The rule in English: 25 degrees or more is T-shirt weather; 15 or more, a light jacket; 5 or more, a coat; anything colder, a coat, hat, and gloves.

weather_advice.py

```
1 temperature = 12
2 if temperature >= 25:
3     advice = "T-shirt weather"
4 elif temperature >= 15:
5     advice = "Bring a light jacket"
6 elif temperature >= 5:
7     advice = "Wear a coat"
8 else:
9     advice = "Wear a coat, hat, and gloves"
10 print(f"{temperature} degrees: {advice}")
```

12 degrees: Wear a coat

Line	Condition	Result	What happens
2	12 >= 25	False	Skip to next check
4	12 >= 15	False	Skip to next check
6	12 >= 5	True	Run line 7, done
8	else never reached		

Line 4 does not need `and temperature < 25`. Python only reaches an **elif** when every condition above it was **False**, so by line 4 it already knows the temperature is below 25. Each rung of the chain can rely on the rungs above it.

Translating English rules into a chain works the same way every time: one rung per rule, in the order the rules must be checked, with **else** for "anything else".

KEY IDEA

In an **if / elif / else** chain, Python runs the block of the first condition that is **True** and skips the rest. **else** runs when no condition is **True**. Exactly one block runs when there is an **else**.

WATCH OUT

Putting the rungs in the wrong order. Since the first true condition wins, a broad condition placed early hides the narrower ones below it:

advice_wrong_order.py

```
1 temperature = 30
2 if temperature >= 5:
3     advice = "Wear a coat"
4 elif temperature >= 15:
5     advice = "Bring a light jacket"
6 elif temperature >= 25:
7     advice = "T-shirt weather"
8 else:
9     advice = "Wear a coat, hat, and gloves"
10 print(f"{temperature} degrees: {advice}")
```

30 degrees: Wear a coat

30 is at least 5, so the first rung is `True` and the chain stops there. The T-shirt rung can never run: any temperature that reaches it would already have matched `>= 5`. When rungs use `>=`, start with the highest threshold and work down.

PAUSE AND TRY

Question 7 Tracing

What does this program print? Watch the boundary value.

shipping.py

```
1 weight = 5
2 if weight > 20:
3     cost = 25
4 elif weight > 5:
5     cost = 12
6 elif weight > 2:
7     cost = 8
8 else:
9     cost = 5
10 print("Cost:", cost)
```

4.7 Truthy and Falsy Values

A program asks for a nickname, and the user just presses Enter. `input()` returns the empty string `""`. You could write `if nickname == ""`, but Python has a shorter way, because a condition does not have to be a bool.

Any value can be used as a condition. Python treats some values as false and all the rest as true. A value that acts as `False` in a condition is **falsey**; a value that acts as `True` is **truthy**. The falsy values are the "empty" or "nothing" ones:

- `False` and `None`
- zero: `0` and `0.0`
- the empty string `""`

- `[]`, an empty list, which you meet on Day 6

Everything else is truthy: any non-zero number (negative ones too), and any string with at least one character, even `" "`, `"0"`, or `"False"`. The built-in function `bool()` shows how Python treats a value:

```
>>> bool(0)
False
>>> bool(0.0)
False
>>> bool("")
False
>>> bool(None)
False
>>> bool([])
False
>>> bool(42)
True
>>> bool(-1)
True
>>> bool("0")
True
>>> bool(" ")
True
```

`"0"` is a one-character string, not the number zero, so it is truthy. `" "` holds a space, so it is not empty.

WORKED EXAMPLE Greeting a user who typed nothing

This run presses Enter without typing anything:

nickname.py

```
1 nickname = input("Nickname: ").strip()
2 if nickname:
3     print(f"Hi, {nickname}!")
4 else:
5     print("Hi, friend!")
```

```
Nickname:
Hi, friend!
```

`strip()` turns empty or all-space input into `""`. `if nickname:` asks "is `nickname` truthy?", meaning "is it non-empty?". It is empty, so the `else` block runs. Had the user typed a nickname, it would be truthy and the `if` block would greet them by name.

not works with truthiness too: `if not nickname:` reads "if there is no nickname".

KEY IDEA

In a condition, `False`, `None`, `0`, `0.0`, `""`, and an empty list are falsy; every other value is truthy. `if text:` means "if `text` is not empty".

WATCH OUT

`answer == "yes" or "y"`. In English, "if the answer is yes or y" is clear. In Python, `or` joins two whole conditions:

yes_or_y.py

```
1 answer = input("Continue? ").strip().lower()
2 if answer == "yes" or "y":
3     print("Continuing...")
4 else:
5     print("Stopping.")
```

```
Continue? no
Continuing...
```

The user typed `no`, yet the program continues. Python reads line 2 as `(answer == "yes") or ("y")`. The left side is `False`, so `or` looks at the right side, `"y"`, which is a non-empty string and therefore `truthy`. When `or` meets a value that is not a `bool`, it gives back one of its two values rather than `True`: here, `"y"` itself. `if` treats that `truthy` string as `true`, so the condition holds for **every** answer. Write the comparison out on both sides of `or`:

yes_or_y_fixed.py

```
1 answer = input("Continue? ").strip().lower()
2 if answer == "yes" or answer == "y":
3     print("Continuing...")
4 else:
5     print("Stopping.")
```

```
Continue? no
Stopping.
```

PAUSE AND TRY**Question 8** Tracing

What does this program print?

predict_truthy.py

```
1 count = 0
2 name = "0"
3 note = None
4 if count:
5     print("A")
6 if name:
7     print("B")
8 if not note:
9     print("C")
10 if count == 0 and name:
11     print("D")
```

4.8 Equality Versus Identity

`None` means "no value yet" (Day 2). Checking for it with `if not value:` is risky, because `0` and `""` are falsy too, and those are real values. Python has an operator that asks precisely "is this `None`?"

Every value in a running program is an **object**: a thing stored in the computer's memory. A variable name refers to an object. Two different objects can hold equal values, like two printed copies of the same book: same words, different copies. Python has two different questions:

- `==` asks about **equality**: do the two values have the same contents? Two copies of the same book are equal.
- `is` asks about **identity**: are both sides the very same object? Two copies are not the same copy.

There is only ever one `None` object in a program, so `value is None` is a precise test for "is this `None`?", and `is not` asks the opposite. This is the standard way to check for `None`:

```
>>> result = None
>>> result is None
True
>>> result is not None
False
>>> result = 0
>>> result is None
False
>>> result is not None
True
```

When `result` is `0`, `result is None` is `False`: `0` is a real value, not "no value".

WORKED EXAMPLE A discount of zero

A discount starts as `None` until one is set, and here it has been set to `0`.

discount_check.py

```
1 discount = 0
2 if discount:
3     print("Truthy check: discount is", discount)
4 if discount is not None:
5     print("None check: discount is", discount)
```

```
None check: discount is 0
```

`if discount:` treats `0` as falsy and skips the line, as if no discount existed.

`if discount is not None:` asks the precise question, "has a discount been set?", and the answer is yes: it is `0`.

For everything except `None`, compare values with `==`. Whether Python keeps one object or several for equal numbers or equal strings is up to Python. It can differ between versions, and even between the REPL and a script, so `is` with numbers or strings can give a different answer from one run to the next.

KEY IDEA

`==` asks whether two values are equal; `is` asks whether they are the same object. Use `is None` and `is not None` to check for `None`, and `==` for everything else.

WATCH OUT

Using `is` to compare text or numbers. `if command is "quit":` or `if count is 10:` asks about objects, not values. It may seem to work in a quick test and then fail in a real program, because nothing promises that equal strings or numbers are the same object. Always write `command == "quit"`.

PAUSE AND TRY**Question 9** Multiple choice

`middle_name` is `None` when a person has no middle name. Which condition is the standard way to check for that?

- A. `middle_name == ""`
- B. `middle_name is None`
- C. `not middle_name`
- D. `middle_name is "None"`

4.9 Conditional Expressions

To print "1 ticket" but "3 tickets", you could write an `if / else` that assigns one of two words:

```
1 if tickets == 1:
2     word = "ticket"
3 else:
4     word = "tickets"
```

Four lines to pick one of two values. Python has a one-line form for this job.

A **conditional expression** picks one of two values based on a condition:

```
1 value_if_true if condition else value_if_false
```

Python works out the condition first. If it is truthy, the whole expression gives `value_if_true`; otherwise it gives `value_if_false`. It reads almost like English: `"ticket"` if there is one ticket, else `"tickets"`. Because it is an **expression** (it gives a value), it can go anywhere a value can: on the right of `=`, inside `print`, or inside an f-string.

WORKED EXAMPLE Singular or plural

plural.py

```

1 tickets = 1
2 word = "ticket" if tickets == 1 else "tickets"
3 print(f"You bought {tickets} {word}.")
4 tickets = 3
5 word = "ticket" if tickets == 1 else "tickets"
6 print(f"You bought {tickets} {word}.")
7 number = 7
8 print(number, "is", "even" if number % 2 == 0 else "odd")

```

```

You bought 1 ticket.
You bought 3 tickets.
7 is odd

```

Line 2: `1 == 1` is True, so word is "ticket". Line 5 is the same expression after `tickets` changed to 3: `3 == 1` is False, so word is "tickets". Line 8 puts a conditional expression straight into `print`: `7 % 2` is 1, not 0, so it gives "odd".

Form	Is a...	Use it to
<code>if / else</code> statement	Statement	Run different blocks of code
<code>a if condition else b</code>	Expression	Pick one of two values

Keep conditional expressions short. When either side needs more than a value, or there are more than two choices, an `if` chain is clearer.

KEY IDEA

`a if condition else b` gives `a` when the condition is truthy and `b` otherwise. It is a value, so it fits inside assignments, `print`, and f-strings. Both the `if` and the `else` are required.

WATCH OUT

Leaving out the `else`. An expression must always give a value, so it needs an answer for the false case too:

missing_else.py

```

1 tickets = 3
2 word = "tickets" if tickets > 1
3 print(word)

```

```

File ".../day04/missing_else.py", line 2
    word = "tickets" if tickets > 1
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

```

SyntaxError: expected 'else' after 'if' expression

Add the false value: `word = "tickets" if tickets > 1 else "ticket".`

PAUSE AND TRY**Question 10** Tracing

Predict exactly what the REPL prints for each line.

```
>>> score = 49
>>> "pass" if score >= 50 else "fail"
>>> score = score + 1
>>> "pass" if score >= 50 else "fail"
>>> items = 0
>>> f"{items} item" + (" " if items == 1 else "s")
```

4.10 A First Look at match

A program that responds to commands often checks one value against a list of fixed choices: `if light == "green":`, `elif light == "yellow":`, `elif light == "red":`, and so on. Each rung repeats `light ==`. Python has a statement designed for exactly this shape.

A **match statement** compares one value, the **subject**, against a series of **case** clauses. Each **case** names a **pattern**; today every pattern is a **literal**, a fixed value written straight into the code, such as `"green"` or `3`. Python checks the cases from the top down, runs the block of the **first** case whose value equals the subject, and skips the rest, just like an if chain.

WORKED EXAMPLE Traffic lights

traffic_light.py

```
1 light = "yellow"
2 match light:
3     case "green":
4         print("Go")
5     case "yellow":
6         print("Slow down")
7     case "red":
8         print("Stop")
9     case _:
10        print("Unknown light")
```

Slow down

- Line 2 names the subject, `light`, and ends with a colon. The `case` lines are indented under it, and each case's block is indented one more level.
- `"yellow" == "green"` is `False`, so line 3's case is skipped.
- Line 5's case matches, so line 6 runs, and the remaining cases are skipped.
- `case _:` uses the **wildcard** `_`, which matches anything. Put it last, as the "anything else" case; it plays the role of `else`.

This does exactly what an if chain with `light == "green"`, `light == "yellow"`, and `light == "red"` would do, but the subject is written once.

To let several values share one block, join them with `|`, read "or":

library_hours.py

```
1 day = "sun"
2 match day:
3     case "sat" | "sun":
4         print("Weekend: open 10:00 to 16:00")
5     case "fri":
6         print("Friday: open 8:00 to 17:00")
7     case _:
8         print("Weekday: open 8:00 to 20:00")
```

```
Weekend: open 10:00 to 16:00
```

"sun" does not equal "sat", but it equals "sun", so the first case matches. `match` can do much more than compare with fixed values; for now, use it when one value is compared against several literals, and use an `if` chain for anything else, such as ranges with `<` or `>`.

KEY IDEA

`match subject:` runs the first case whose literal equals the subject. `|` lets one case accept several values, and `case _:` matches anything, so it goes last.

WATCH OUT

Expecting a near match. A literal case matches only an equal value, and equality is case-sensitive:

match_case_sensitive.py

```
1 command = "Help"
2 match command:
3     case "help":
4         print("Commands: start, stop, help")
5     case "start":
6         print("Starting...")
7 print("Done.")
```

```
Done.
```

"Help" does not equal "help", no case matches, and there is no `case _:`, so the `match` does nothing at all, with no error. Clean the subject first, as in `match command.lower():`, and add a `case _:` that reports unknown values.

PAUSE AND TRY

Question 11 Tracing

What does this program print?

shirt_price.py

```

1 size = "m"
2 match size:
3     case "S" | "XS":
4         price = 12
5     case "M" | "L":
6         price = 15
7     case _:
8         price = 18
9 print("Size", size, "costs", price)

```

Putting it together

A cinema charges 6.00 for children under 12, 7.00 for seniors 65 and over, and 10.00 for everyone else. On Tuesdays every ticket is 2.00 off, and on Fridays teenagers from 13 to 17 get free popcorn. This run types 15 and Friday:

cinema.py

```

1 age = int(input("Age: "))
2 day = input("Day: ").strip().lower()
3
4 if age < 12:
5     ticket = "child"
6     price = 6.0
7 elif age >= 65:
8     ticket = "senior"
9     price = 7.0
10 else:
11     ticket = "standard"
12     price = 10.0
13
14 discount = 2.0 if day == "tuesday" else 0.0
15 price = price - discount
16 print(f"Ticket: {ticket}, {price:.2f}")
17
18 if day == "friday" and 13 <= age <= 17:
19     print("Teen Friday: free popcorn!")

```

```

Age: 15
Day: Friday
Ticket: standard, 10.00
Teen Friday: free popcorn!

```

Line	Worked out	Result	Idea
1	int("15")	15	Day 3
2	"Friday".strip().lower()	"friday"	Day 3
4	15 < 12	False	4.1, 4.6

7	<code>15 >= 65</code>	<code>False</code>	4.6
10	<code>else</code> block sets <code>ticket, price</code>	<code>"standard", 10.0</code>	4.6
14	<code>"friday" == "tuesday"</code> is <code>False</code>	<code>0.0</code>	4.9
15	<code>10.0 - 0.0</code>	<code>10.0</code>	Day 2
18	<code>True</code> and <code>13 <= 15 <= 17</code>	<code>True</code>	4.2, 4.3

The if chain gives every customer exactly one ticket type, with the narrow cases (under 12, 65 and over) checked before `else` catches the rest. Line 14 uses a conditional expression because it only picks one of two values. Line 18 joins a string check and a chained range check with `and`. Line 2's `lower()` is what makes Friday match `"friday"`.

Vocabulary

Term	Definition	Example
comparison operator	<code>==</code> , <code>!=</code> , <code><</code> , <code>></code> , <code><=</code> , <code>>=</code> : compares two values and gives a bool	<code>score >= 50</code>
condition	An expression whose truth decides what runs	<code>age < 12</code>
chained comparison	Several comparisons in a row; true when every link is true	<code>13 <= age <= 19</code>
logical operator	<code>and</code> , <code>or</code> , <code>not</code> : combine or flip conditions	<code>tall</code> and <code>old_enough</code>
truth table	A table of a logical operator's result for every combination	<code>True</code> and <code>False</code> is <code>False</code>
short-circuit evaluation	Skipping the right side of <code>and</code> / <code>or</code> when the left decides the answer	<code>count > 0</code> and <code>total / count</code>
if statement	Runs its block only when its condition is true	<code>if score > 90:</code>
block	The indented lines that belong to an <code>if</code> , <code>else</code> , or <code>case</code>	4 spaces in
selection	Choosing which code runs based on a condition	an if chain
else / elif	The block when nothing above matched / another condition in a chain	<code>elif score >= 80:</code>
if chain	<code>if</code> , any <code>elif</code> s, and an optional <code>else</code> ; the first true rung runs	grade from score
truthy / falsy	Acts as <code>True</code> / as <code>False</code> in a condition	<code>"0"</code> / <code>""</code>
object	A value stored in memory; names refer to objects	the one <code>None</code>
equality / identity	Same value (<code>==</code>) / the same object (<code>is</code>)	<code>x is None</code>
conditional expression	Picks one of two values based on a condition	<code>"odd" if n % 2 == 1 else "even"</code>

match statement	Compares a subject against case patterns and runs the first that fits	match command:
literal	A fixed value written into code	"help", 3
wildcard	The pattern <code>_</code> , which matches anything	case <code>_</code> :

Summary

- **Comparison operators** give `True` or `False`; arithmetic happens first, and strings compare alphabetically with capitals first. A **chained comparison** like `low <= value <= high` checks a range.
- `and` needs both sides true, `or` at least one, and `not` flips. `and` binds tighter than `or`, and **short-circuit evaluation** skips the right side when the left decides, so put safety checks first.
- An **if chain** runs the block of the first true condition and skips the rest; `else` catches everything else. Indentation marks each **block**.
- `False`, `None`, `0`, `0.0`, `""`, and an empty list are **falsy**; everything else is **truthy**.
- `==` compares values; `is` compares objects. Use `is None` for `None` and `==` for everything else.
- A **conditional expression** `a if condition else b` picks a value, and `match / case` compares one value against fixed choices.

Practice

Do these in your `day04` folder. Type each program yourself, run it, and only then open the answer. Run each one several times with different typed input, including values right on a boundary.

Question 12 Short answer

Letter grade. Ask for a numeric score and print a letter grade: 90 and above is A, 80 to 89.9 is B, 70 to 79.9 is C, 60 to 69.9 is D, and below 60 is F. If the score is not between 0 and 100, print a message instead of a grade.

Question 13 Short answer

Leap year. A year is a leap year if it is divisible by 4, except that years divisible by 100 are not, except that years divisible by 400 are. Ask for a year and print whether it is a leap year. (A number is divisible by 4 when `year % 4 == 0`.)

Question 14 Short answer

Text menu. Show the user three commands, read one with `input()`, and print a different response for each. For anything else, print a message that names the unknown command. The program handles one command per run; Day 5 shows how to keep asking.

Review questions

Question 15 Tracing

What does this program print?

cart.py

```
1 items_in_cart = 3
2 coupon = ""
3 total = 42.0
4 if items_in_cart > 5 or total > 40 and not coupon:
5     print("Free delivery")
6 else:
7     print("Delivery: 4.99")
8 if coupon:
9     print("Coupon:", coupon)
10 size = "small" if items_in_cart < 5 else "large"
11 print("Order size:", size)
```

Question 16 Multiple choice

What does this program print?

color_check.py

```
1 color = "green"
2 if color == "red" or "blue":
3     print("Warm or cool")
4 else:
5     print("Something else")
```

- A. Warm or cool
- B. Something else
- C. Nothing
- D. A SyntaxError

Question 17 Debugging

This program should print `Not hot` and then `Checked the temperature`. It stops before printing anything, with `SyntaxError: invalid syntax` pointing at `else`. Explain the error and write the fix.

misplaced_else.py

```
1 temperature = 18
2 if temperature > 25:
3     print("Hot")
4 print("Checked the temperature.")
5 else:
6     print("Not hot")
```

Question 18 Short answer

A game stores `high_score = None` until the first game is played. Here, the first game scored 0. Explain what this program prints and why the second `if` is the better check.

high_score.py

```
1 high_score = 0
2 if not high_score:
3     print("Truthy check: no games played yet")
4 if high_score is None:
5     print("None check: no games played yet")
6 else:
7     print("None check: high score is", high_score)
```

Question 19 Tracing

What does this program print?

command_status.py

```
1 command = " STOP "
2 match command.strip().lower():
3     case "go" | "start":
4         status = "running"
5     case "stop" | "halt":
6         status = "stopped"
7     case _:
8         status = "unknown"
9 symbol = ">" if status == "running" else "#"
10 print(symbol, status)
```

Go further

The official Python Tutorial covers today's statements in [More Control Flow Tools: if Statements](#) and [More Control Flow Tools: match Statements](#). The match section goes well beyond literal patterns; read its first examples now and the rest later.

Reflection

Check each statement. If the answer is "not yet," reread the linked subsection.

- ☐ I can compare numbers and strings and predict the result. Reread: [4.1 Comparison Operators](#)
- ☐ I can write a range check as a chained comparison. Reread: [4.2 Chained Comparisons](#)
- ☐ I can combine conditions with and, or, and not, and add parentheses where needed. Reread: [4.3 Combining Conditions: and, or, not](#)
- ☐ I can explain short-circuit evaluation and use it to guard a division. Reread: [4.4 Short-Circuit Evaluation](#)
- ☐ I can write an if statement with a correctly indented block. Reread: [4.5 if Statements and Blocks](#)
- ☐ I can translate English rules into an if, elif, and else chain. Reread: [4.6 else and elif](#)
- ☐ I can say which values are truthy and which are falsy. Reread: [4.7 Truthy and Falsy Values](#)
- ☐ I can check for None with is and compare values with ==. Reread: [4.8 Equality Versus Identity](#)
- ☐ I can pick one of two values with a conditional expression. Reread: [4.9 Conditional Expressions](#)
- ☐ I can match a value against fixed choices with match and case. Reread: [4.10 A First Look at match](#)

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

```
>>> price = 4.5
>>> price >= 4.5
True
>>> price * 2 != 9
False
>>> "Dog" == "dog"
False
>>> "ant" < "Bee"
False
>>> 10 == 10.0
True
```

- `4.5 >= 4.5` is `True`: equal counts.
- Arithmetic first: `price * 2` is `9.0`, and `9.0 != 9` asks "are they different?" They are equal, so `False`.
- Comparisons are case-sensitive: `D` and `d` differ, so `False`.
- Capitals come before small letters, so `"Bee"` comes before `"ant"`, and `"ant" < "Bee"` is `False`.
- An int and a float with the same value are equal: `True`.

Reread 4.1 *Comparison Operators* if any answer differed.

Question 2

B. Both links must hold: `0 < minutes` lets in 1 and up, and `minutes < 60` lets in up to 59.

- **A** is wrong: `<` leaves out the ends, so 1 and 59 fail.
- **C** is wrong: its links are `minutes > 0` and `0 < 60`; the second is always true, so 100 passes.
- **D** is wrong: reading right to left it is `1 < minutes < 60`, which leaves out 1.

Reread 4.2 *Chained Comparisons* if you chose differently.

Question 3

```
>>> x = 7
>>> x > 5 and x < 10
True
>>> x < 5 or x > 10
False
>>> not x == 7
False
>>> x > 5 or x > 100 and x < 0
True
```

- `True and True` is `True`.
- `False or False` is `False`: 7 is neither below 5 nor above 10.
- `x == 7` is worked out first, giving `True`, and `not` flips it: `False`.

- `and` binds tighter, so this means `x > 5` or `(x > 100 and x < 0)`. Python works left to right: `x > 5` is `True`, so `or` stops there and gives `True`. The `and` part never runs (4.4 explains why).

Reread 4.3 *Combining Conditions: and, or, not* if any answer differed.

Question 4

Model answer: in the first, `count != 0` runs first and is `False` when `count` is 0, so `and` stops and gives `False` without dividing. In the second, the division runs first and stops the program with a `ZeroDivisionError`. When `count` is 4, nothing divides by zero, both sides run in both versions, and both give the same answer, because `and` gives the same result in either order when both sides can be worked out.

A correct answer says that Python works out the left side first, that `and` skips the right side after a `False`, and that the order matters only when one side can fail.

Reread 4.4 *Short-Circuit Evaluation* if your reasoning differed.

Question 5

```
Looking at the shoes...
Buying them.
Saving up instead.
Money left: 5
```

Line	Worked out	money
4	<code>20 >= 15: True</code>	20
6	<code>20 - 15</code>	5
7	<code>5 < 15: True</code>	5

The two `if` statements are separate questions. The first block buys the shoes and lowers `money` to 5. Then the second `if` asks its own question with the **new** value, `5 < 15`, and also runs. The program buys the shoes and then says it is saving up. Section 4.6 shows how to make the two blocks exclusive.

Reread 4.5 *if Statements and Blocks* if your output differed.

Question 6

```
SyntaxError: expected ':'
```

Line 2 starts an `if` statement but has no colon at the end, so Python does not know where the condition stops and the block begins. Because it is a syntax error, nothing runs, not even line 1. Add the colon:

library_fine_fixed.py

```
1 days_late = 4
2 if days_late > 0:
3     print("Fine:", days_late * 0.25)
4     print("Thank you!")
```

```
Fine: 1.0
Thank you!
```

Reread 4.5 *if Statements and Blocks* if you missed it.

Question 7

Cost: 8

- `5 > 20` is `False`.
- `5 > 5` is `False`: 5 is not **greater** than 5.
- `5 > 2` is `True`, so `cost` is 8 and the chain stops.

If you answered 12, check the difference between `>` and `>=`: a parcel of exactly 5 is not over 5.

Reread 4.6 *else and elif* if your output differed.

Question 8

B
C
D

Line	Condition	Truthy?	Prints
4	<code>0</code>	No	
6	<code>"0"</code>	Yes	B
8	<code>not None</code>	Yes	C
10	<code>True and "0"</code>	Yes	D

`count` is the number 0, which is falsy, but `name` is the **string** `"0"`, which has one character and is truthy. `None` is falsy, so `not None` is true.

Reread 4.7 *Truthy and Falsy Values* if your output differed.

Question 9

B. There is only one `None` object, so `is None` checks for it exactly.

- **A** is wrong: `None` is not the empty string, so this is `False` when `middle_name` is `None`.
- **C** gives the right answer for `None`, but it is also true for `""`, so it cannot tell "no value" from "an empty value".
- **D** is wrong: `"None"` in quotes is a four-letter string, not the value `None`, and `is` should not compare strings anyway.

Reread 4.8 *Equality Versus Identity* if you chose differently.

Question 10

```
>>> score = 49
>>> "pass" if score >= 50 else "fail"
'fail'
>>> score = score + 1
>>> "pass" if score >= 50 else "fail"
'pass'
>>> items = 0
>>> f"{items} item" + ("s" if items == 1 else "s")
'0 items'
```

- `49 >= 50` is `False`, so the expression gives `'fail'`.
- `score` becomes 50, and `50 >= 50` is `True`: `'pass'`.
- `0 == 1` is `False`, so the parentheses give `"s"`, and `"0 item" + "s"` is `'0 items'`. (English uses the plural for zero, which this handles.)

Reread 4.9 *Conditional Expressions* if any answer differed.

Question 11

Size `m` costs 18

The subject is a small `"m"`. The first case accepts `"s"` or `"xs"` and the second `"M"` or `"L"`, all capitals, so neither equals `"m"`. Only the wildcard `_` matches, and `price` is 18. If you answered 15, remember that matching is case-sensitive.

Reread 4.10 *A First Look at match* if your output differed.

Question 12

One possible solution:

grade.py

```
1 score = float(input("Score: "))
2 if not 0 <= score <= 100:
3     print("Scores run from 0 to 100.")
4 elif score >= 90:
5     print("Grade: A")
6 elif score >= 80:
7     print("Grade: B")
8 elif score >= 70:
9     print("Grade: C")
10 elif score >= 60:
11     print("Grade: D")
12 else:
13     print("Grade: F")
```

Score: 89.5
Grade: B

`float` allows scores like 89.5. The first rung uses a chained comparison: `not 0 <= score <= 100` is true for anything outside the range. The other rungs run highest first, so each only needs a lower bound: by the time Python reaches `score >= 80`, it already knows the score is below 90. 89.5 fails `>= 90` and passes `>= 80`, so it gets a B. Try 90, 60, 59.9, and 101 too.

Reread 4.6 *else and elif* and 4.2 *Chained Comparisons* if your result differed.

Question 13

One possible solution:

leap_year.py

```
1 year = int(input("Year: "))
2 if year % 400 == 0:
3     is_leap = True
4 elif year % 100 == 0:
5     is_leap = False
6 elif year % 4 == 0:
7     is_leap = True
8 else:
9     is_leap = False
10
11 if is_leap:
12     print(f"{year} is a leap year.")
13 else:
14     print(f"{year} is not a leap year.")
```

```
Year: 1900
1900 is not a leap year.
```

The English rule has exceptions to exceptions, so the chain checks the **most specific** rule first. 1900 is not divisible by 400, but it is divisible by 100, so the second rung sets `is_leap` to `False`. Checking `% 4` first would wrongly call 1900 a leap year. The last `if` uses the bool directly as its condition.

The same rule also fits in one condition, using `and`, `or`, and parentheses:

```
>>> year = 2024
>>> year % 4 == 0 and (year % 100 != 0 or year % 400 == 0)
True
>>> year = 1900
>>> year % 4 == 0 and (year % 100 != 0 or year % 400 == 0)
False
>>> year = 2000
>>> year % 4 == 0 and (year % 100 != 0 or year % 400 == 0)
True
>>> year = 2023
>>> year % 4 == 0 and (year % 100 != 0 or year % 400 == 0)
False
```

Read it as "divisible by 4, and either not a century or divisible by 400". Either version is correct; the if chain is easier to check against the English rule.

Reread 4.6 *else and elif* and 4.3 *Combining Conditions: and, or, not* if your result differed.

Question 14

One possible solution, for a café:

menu.py

```

1 print("Commands: hours, price, help")
2 command = input("Command: ").strip().lower()
3 match command:
4     case "hours":
5         print("Open 8:00 to 18:00, Monday to Saturday.")
6     case "price":
7         print("A coffee costs 3.50.")
8     case "help":
9         print("Type hours, price, or help.")
10    case _:
11        print(f"Unknown command: {command}")

```

Four runs, typing `Hours`, `price`, `help`, and `coffee`:

```

Commands: hours, price, help
Command: Hours
Open 8:00 to 18:00, Monday to Saturday.

```

```

Commands: hours, price, help
Command: price
A coffee costs 3.50.

```

```

Commands: hours, price, help
Command: help
Type hours, price, or help.

```

```

Commands: hours, price, help
Command: coffee
Unknown command: coffee

```

`strip().lower()` cleans the input, so `Hours` matches `"hours"`. Each command is a fixed value, which is what `match` is for, and `case _:` catches everything else. An `if` chain with `command == "hours"` and so on works just as well.

Reread 4.10 *A First Look at match* if your result differed.

Question 15

```

Free delivery
Order size: small

```

Line	Worked out	Result
4	Left side of <code>or</code> : <code>3 > 5</code>	False
4	Right side: <code>42.0 > 40</code> and <code>not ""</code> is <code>True</code> and <code>True</code>	True
4	<code>False or True</code>	True
8	<code>""</code> is falsy	skipped
10	<code>3 < 5</code>	"small"

Because `and` binds tighter than `or`, line 4 means

`items_in_cart > 5 or (total > 40 and not coupon)`. Python works it out left to right: the left side of `or` is `False`, so it goes on to the `and` part. The empty coupon is falsy, so `not coupon` is `True`, and line 9 never runs.

Reread 4.3 *Combining Conditions: and, or, not* and 4.7 *Truthy and Falsy Values* if your output differed.

Question 16

A. Python reads the condition as `(color == "red") or ("blue")`. The comparison is `False`, so `or` gives the right side, `"blue"`, a non-empty string, which is `truthy`.

- **B** is what the programmer intended; it needs `color == "red" or color == "blue"`.
- **C** is wrong: with an `else`, one of the two blocks always runs.
- **D** is wrong: the code is valid Python; it just means something other than it seems.

Reread the Watch Out in 4.7 *Truthy and Falsy Values* if you chose differently.

Question 17

Line 4 is back at the left edge, so it ends the `if` statement. When Python reaches `else` on line 5, there is no `if` directly above it for it to belong to, because an `else` must come straight after its `if` block. Move the line that should always run below the whole `if / else`:

`misplaced_else_fixed.py`

```
1 temperature = 18
2 if temperature > 25:
3     print("Hot")
4 else:
5     print("Not hot")
6 print("Checked the temperature.")
```

```
Not hot
Checked the temperature.
```

Reread 4.5 *if Statements and Blocks* and 4.6 *else and elif* if you missed it.

Question 18

```
Truthy check: no games played yet
None check: high score is 0
```

Model answer: 0 is falsy, so `not high_score` is `True`, and the first check wrongly says no games have been played. `high_score is None` asks only whether the value is `None`, which it is not, so the `else` reports the real score of 0. A correct answer says that 0 is falsy, that is `None` checks for `None` alone, and that a score of 0 is a real value.

Reread 4.7 *Truthy and Falsy Values* and 4.8 *Equality Versus Identity* if your reasoning differed.

Question 19

```
# stopped
```

- The subject is `" STOP ".strip().lower()`, which is `"stop"`.

- The first case accepts "go" or "start": no match. The second accepts "stop" or "halt": match, so status is "stopped".
- Line 9: "stopped" == "running" is False, so symbol is "#".
- print puts a space between its two arguments.

Reread *4.10 A First Look at match* and *4.9 Conditional Expressions* if your output differed.