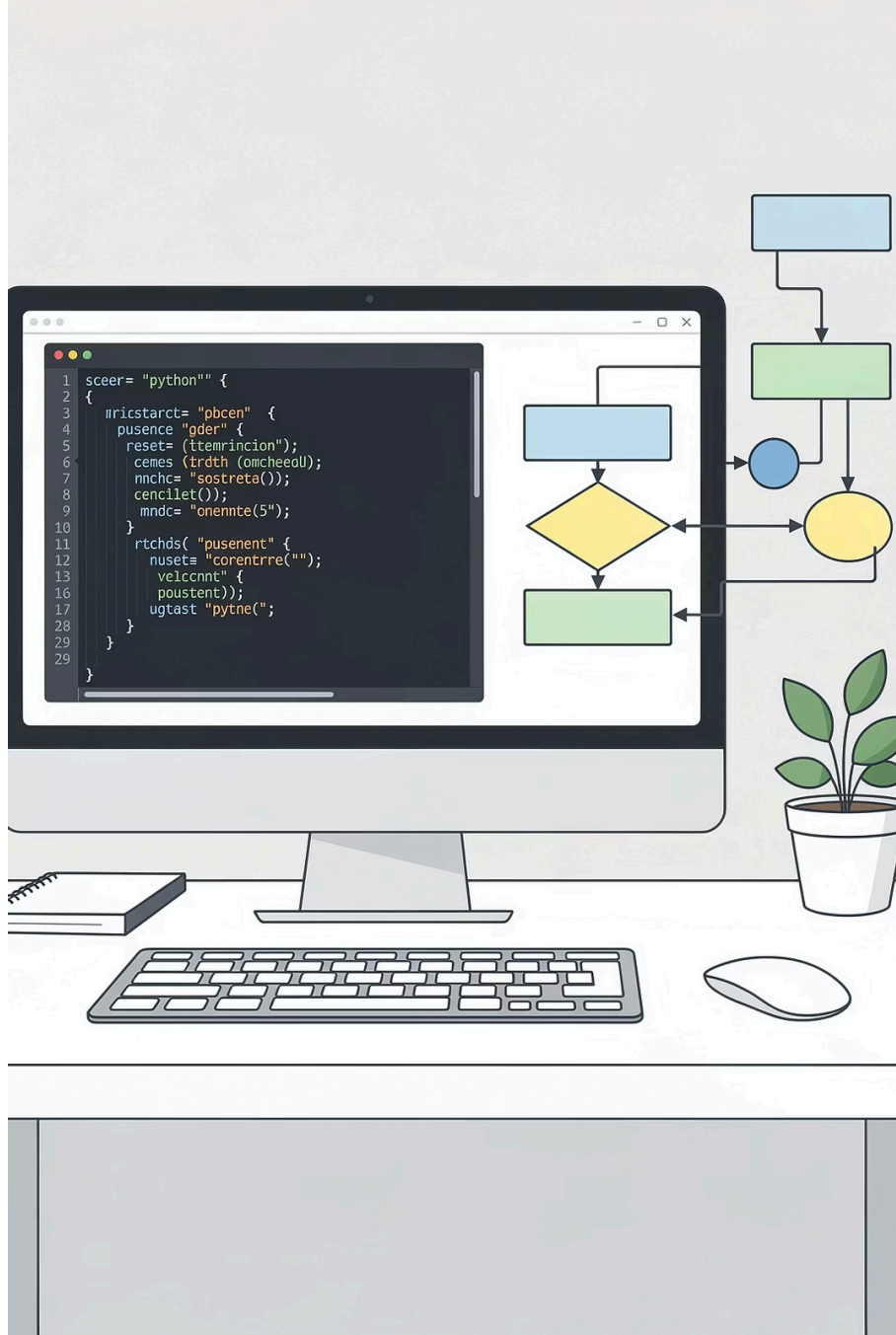
Choosing if, while, or for in Python: Control Flow Made Simple

Master the art of decision-making and repetition in your Python code with the right control structures.



When to Use `if`

Decision-Making Logic

Use `if` statements to make your code intelligent and responsive. They evaluate conditions and execute code only when those conditions are true.

Think of it as a fork in the road: choose one path based on the conditions you encounter.

Common Use Cases

- Validating user input before processing
- Checking if a value meets specific criteria
- Responding differently based on data states
- Implementing conditional logic in functions

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

When to Use `while`

Indefinite Repetition

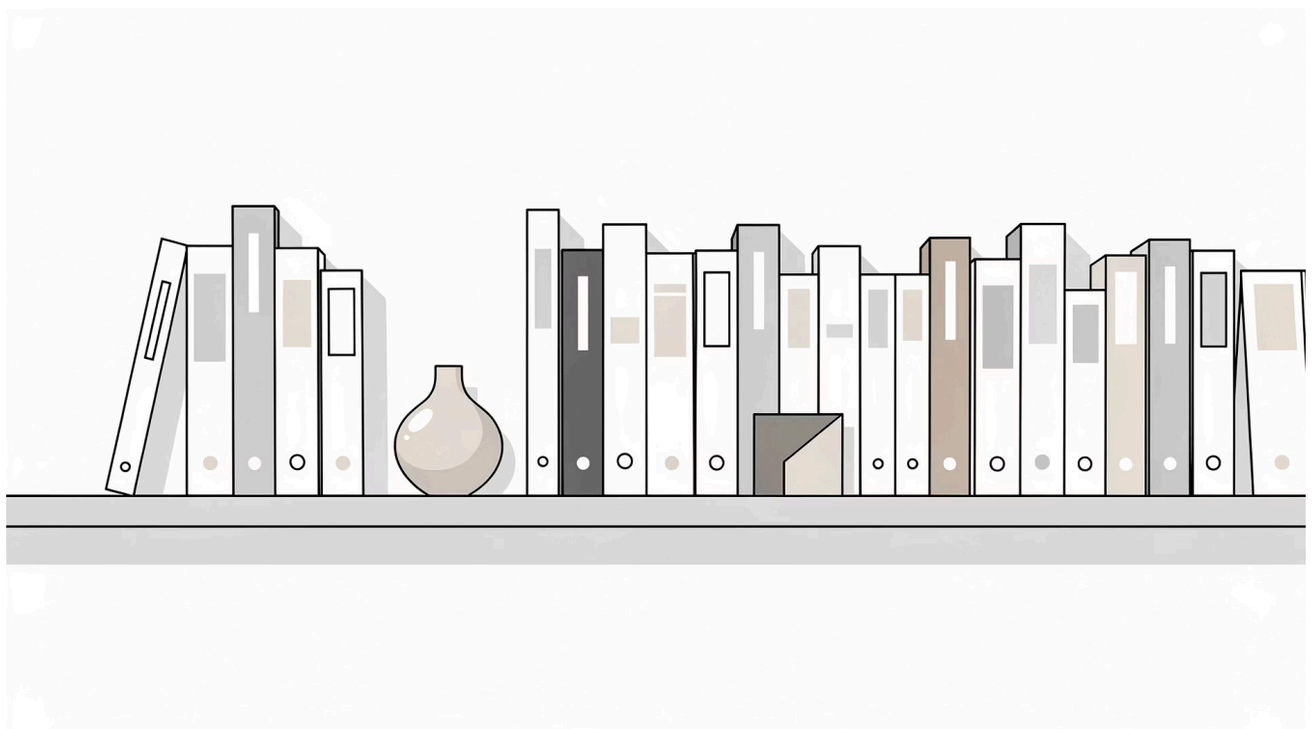
Use `while` loops when you need to repeat actions as long as a condition remains true — perfect when you don't know the exact number of iterations in advance.

Dynamic Conditions

Like a playground swing that keeps going until you decide to stop, `while` loops continue until a condition changes or is explicitly broken.

```
user_input = ""
while user_input != "quit":
 user_input = input("Enter command: ")
 process(user_input)
```

When to Use `for`



Iterate Over Collections

Use `for` loops to iterate over sequences like lists, strings, or ranges. Best when you know how many times to repeat or want to process each item systematically.

Perfect For:

- Processing every element in a list
- Looping through characters in a string
- Executing a fixed number of times
- Accessing indexed collections

```
names = ["Alice", "Bob", "Carol"]
for name in names:
 print(f"Hello, {name}!")
```

The Power of `range()` in `for` Loops

01

Basic Syntax

`range(start, stop, step)` generates a sequence of numbers. The `stop` value is exclusive — counting stops just before it.

02

Simple Range

`for i in range(5)`: loops from 0 to 4, executing exactly 5 times. Default start is 0 and step is 1.

03

Customize Parameters

Specify start, stop, and step values to create custom sequences — including negative steps for counting backward!

04

Practical Applications

Great for counting loops, indexing collections, creating number sequences, or controlling iteration precision.

`range()` Examples

Basic Range

```
for i in range(5):
    print(i)
# Output: 0, 1, 2, 3, 4
```

Default behavior: starts at 0, counts up by 1.

Custom Start and Stop

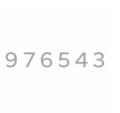
```
for i in range(2, 9):
    print(i)
# Output: 2, 3, 4, 5, 6, 7, 8
```

Specify both start and stop values for custom ranges.

Step Increment

```
for i in range(0, 10, 2):
    print(i)
# Output: 0, 2, 4, 6, 8
```

Use the step parameter to skip values or count by increments.



Counting Backward

```
for i in range(10, 0, -1):
    print(i)
# Output: 10, 9, 8, ..., 1
```

Negative steps allow countdown sequences — perfect for timers!