Python Control Flow & Data Structures

Day 3: Making Decisions and Organizing Data

TODAY'S JOURNEY

What We'll Cover

01

Control Flow Basics

Making decisions with if, elif, and else statements

02

Loops & Iteration

Repeating tasks with for and while loops

03

Loop Control

Managing flow with break, continue, and pass

04

Data Structures

Organizing data with lists, tuples, sets, and dictionaries

05

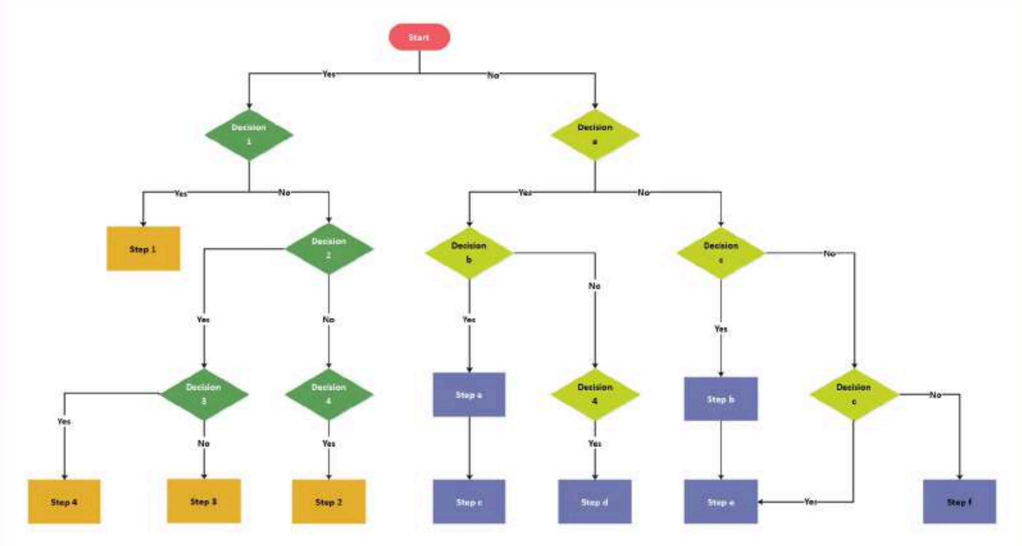
Real-World Practice

Apply concepts through practical examples

Why Control Flow Matters

Programs need to make decisions and repeat actions. Without control flow, every line would execute once, top to bottom—no flexibility, no intelligence.

Real-world analogy: Think of a recipe. You don't just mix everything blindly. You check if water is boiling, repeat stirring until smooth, and decide whether to add more salt based on taste.



Control flow gives your programs the ability to respond to different situations intelligently.

The if Statement: Making Decisions

The `if` statement evaluates a condition. If true, the indented code block runs. If false, it's skipped.

```
age = 20

if age >= 18:
    print("You are an adult")
    print("You can vote")
```

Condition

`age >= 18` evaluates to True or False

Indentation

4 spaces define the code block that runs if true

Colon Required

Don't forget the `:` at the end of the if line

Adding else: Handle Both Cases

Use `else` to specify what happens when the condition is false. No condition needed after else—it catches everything the if didn't.

```
temperature = 30

if temperature > 25:
 print("It's hot outside")
else:
 print("It's comfortable")
```

The elif Ladder: Multiple Conditions

When you need to check multiple conditions in sequence, use `elif` (short for "else if"). Python checks each condition from top to bottom and stops at the first true one.

```
score = 85

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

Only **one** block executes. Once a condition is true, the rest are skipped.

Nested Conditions: Decisions Within Decisions

You can place if statements inside other if statements. This creates layered decision-making logic.

```
has_ticket = True
age = 16

if has_ticket:
    if age >= 18:
        print("Entry allowed")
    else:
        print("Need adult supervision")
else:
    print("Buy a ticket first")
```

Tip: Keep nesting to 2-3 levels maximum. Deeper nesting makes code hard to read and debug.

Use logical operators like `and` and `or` to combine conditions instead when possible.

Comparison & Logical Operators

Comparison

- `==` equal to
- `!=` not equal
- `<` `>` `<=` `>=`

Logical

- `and` both must be true
- `or` at least one true
- `not` reverses the result

Example

```
if age >= 18 and has_id:
  print("Can enter")
```

CHAPTER 2

Loops: Repeating Actions

Automate repetition without copy-pasting code

Why We Need Loops

Imagine printing numbers 1 to 100. Without loops, you'd write 100 print statements. Loops let you repeat actions efficiently with just a few lines of code.

1

Manual

Copy-paste code 100 times—error-prone and tedious

2

Loop

Write once, repeat automatically—clean and maintainable

Loops are fundamental to programming. Master them early!

The for Loop: Iterate Over Sequences

Use for loops when you know how many times to repeat, or when iterating over a collection of items.

```
fruits = ["apple", "banana", "cherry"]

for fruit in fruits:
    print(f"I love {fruit}")
```

How it works

1. Takes the first item from the list
2. Assigns it to the variable fruit
3. Runs the indented code
4. Repeats for each item

The range() Function

The `range()` function generates a sequence of numbers. It's perfect for counting loops.

range(stop)

```
range(5)  # 0, 1, 2, 3, 4
```

Starts at 0, goes up to (but not including) stop

range(start, stop)

```
range(2, 7)  # 2, 3, 4, 5, 6
```

Starts at start, stops before stop

range(start, stop, step)

```
range(0, 10, 2)  # 0, 2, 4, 6, 8
```

Increment by step instead of 1

for Loop with range(): Counting Made Easy

```
# Print numbers 1 to 5
for i in range(1, 6):
    print(i)

# Print even numbers from 0 to 10
for num in range(0, 11, 2):
    print(num)
```

Common use case: When you need to repeat an action a specific number of times, or access items by index.

The while Loop: Condition-Based Repetition

A `while` loop repeats as long as a condition remains true. Use it when you don't know in advance how many iterations you'll need.

```
count = 0

while count < 5:
    print(f"Count is {count}")
    count += 1
```

Warning: Make sure the condition eventually becomes false, or you'll create an infinite loop!

for vs while: When to Use Each

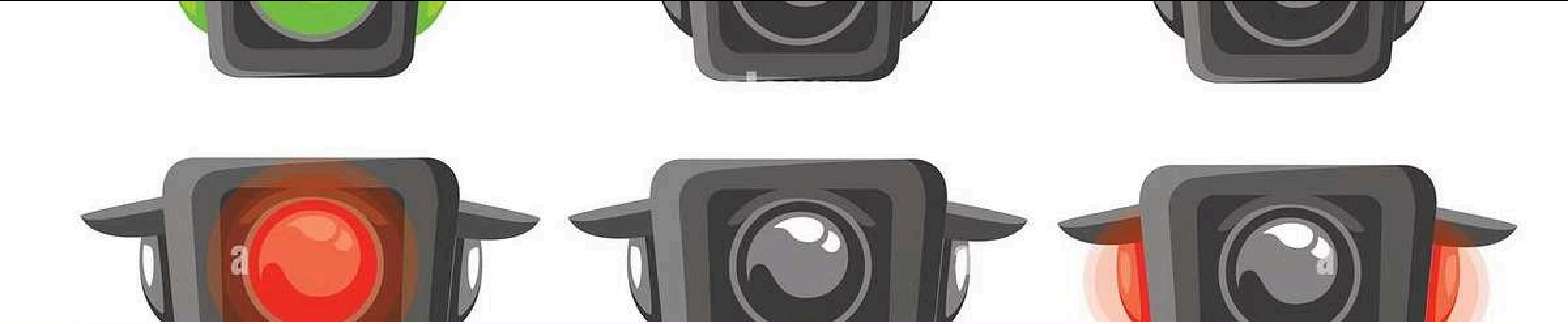
Use for loops when:

- Iterating over a sequence (list, string, range)
- You know the number of iterations
- You need cleaner, more readable code

Use while loops when:

- Repeating until a condition changes
- Number of iterations is unknown
- Waiting for user input or events

Rule of thumb: Prefer `for` loops when possible—they're safer and more Pythonic.



LOOP CONTROL

break, continue, and pass

These keywords give you fine-grained control over loop execution, letting you exit early, skip iterations, or hold a place in your code.

break: Exit the Loop Early

The `break` statement immediately exits the loop, regardless of the loop condition.

```
# Find first number divisible by 7
for num in range(1, 100):
    if num % 7 == 0:
        print(f"Found: {num}")
        break
```

Result: Prints "Found: 7" and stops. Doesn't check 8, 9, 10...

Use `break` when you've found what you're looking for and don't need to continue.

continue: Skip to Next Iteration

The `continue` statement skips the rest of the current iteration and jumps to the next one.

```
# Print odd numbers only
for num in range(10):
    if num % 2 == 0:
        continue
    print(num)
```

This prints: 1, 3, 5, 7, 9. When `num` is even, `continue` skips the print statement.

pass: The Placeholder Statement

`pass` does nothing. It's used as a placeholder when syntax requires a statement but you don't want to execute anything yet.

```
for i in range(5):
    if i == 3:
        pass  # TODO: handle later
    else:
        print(i)
```

Common in development when you're outlining structure before implementing logic. Prevents syntax errors in incomplete code.

Loop Control: Visual Summary

break

Exit loop immediately

Example: Stop searching once item is found

continue

Skip to next iteration

Example: Skip invalid data, process rest

pass

Do nothing (placeholder)

Example: Define structure before implementation

CHAPTER 3

Data Structures

Organizing information efficiently

Why Data Structures?

Real programs handle lots of data: user lists, product inventories, transaction records. You can't create a separate variable for each item—you need structured ways to organize and access data.

Organization

Group related data together logically

Efficiency

Access and modify data quickly

Scalability

Handle growing amounts of data seamlessly

Python's Core Data Structures

Lists

Ordered, mutable collections

Tuples

Ordered, immutable collections

Sets

Unordered, unique elements only

Dictionaries

Key-value pairs for fast lookups

Each has unique characteristics that make it ideal for different situations.

Lists: Flexible Ordered Collections

Lists store multiple items in a single variable. They're ordered (items stay in the order you add them), mutable (you can change them), and allow duplicates.

```
students = ["Alice", "Bob", "Charlie", "Alice"]
numbers = [10, 20, 30, 40, 50]
mixed = [1, "hello", 3.14, True]
```

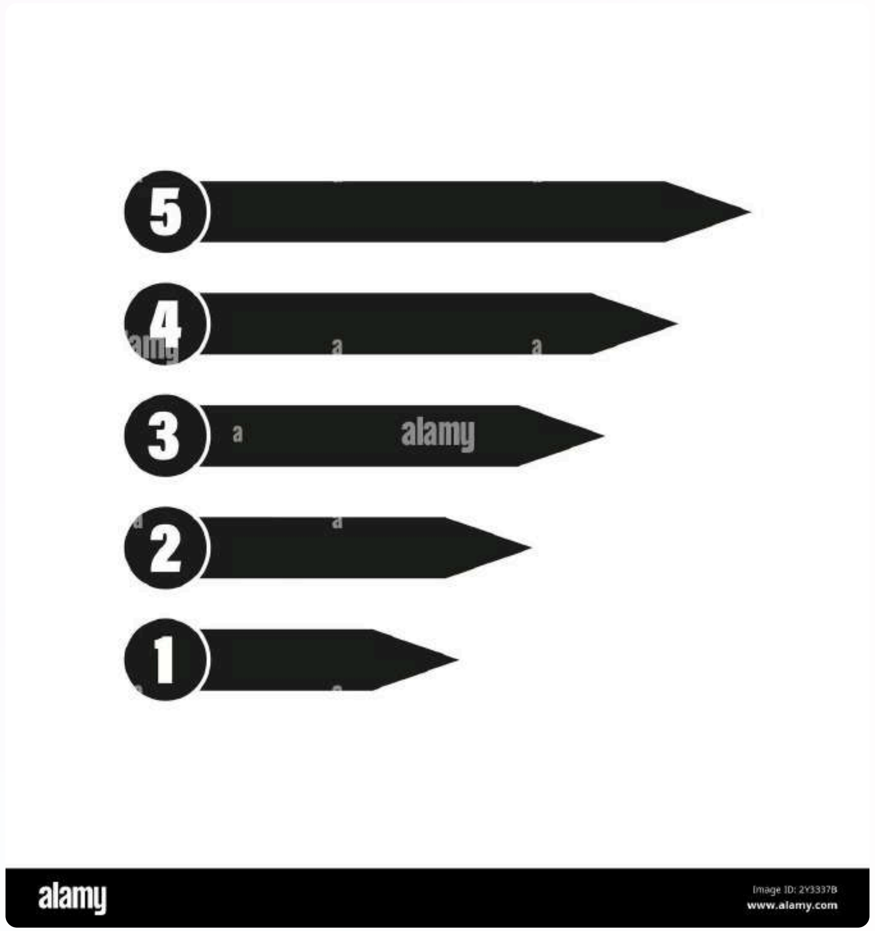
Create a list with square brackets `[]`. Items can be any data type, even mixed types in one list.

Accessing List Elements

Access items by index (position), starting at 0. Use negative indices to count from the end.

```
fruits = ["apple", "banana", "cherry"]

print(fruits[0])   # apple
print(fruits[1])   # banana
print(fruits[-1])  # cherry (last)
print(fruits[-2])  # banana (second-to-last)
```



Modifying Lists

Lists are mutable, meaning you can change, add, or remove items after creation.

Change an item

```
fruits[1] = "blueberry"
```

Add to end

```
fruits.append("orange")
```

Insert at position

```
fruits.insert(1, "mango")
```

Remove item

```
fruits.remove("apple")
```

Remove by index

```
fruits.pop(0)
```

Useful List Operations

Length

```
len(fruits)
```

Returns number of items

Check membership

```
"apple" in fruits
```

Returns True or False

Slicing

```
fruits[1:3]
```

Get subset of items

Sort

```
fruits.sort()
```

Arrange in order

Reverse

```
fruits.reverse()
```

Flip the order

Count occurrences

```
fruits.count("apple")
```

How many times item appears

Tuples: Immutable Sequences

Tuples are like lists but **immutable**—once created, you can't change them. Use parentheses `()` to create tuples.

```
coordinates = (10, 20)
rgb_color = (255, 128, 0)
person = ("Alice", 25, "Engineer")
```

Why use tuples?

- **Data integrity:** Prevents accidental changes
- **Performance:** Slightly faster than lists
- **Dictionary keys:** Tuples can be keys, lists can't

Working with Tuples

You can access tuple elements by index, just like lists. But you cannot modify, add, or remove items.

```
point = (5, 10)
x = point[0]  # Access: OK
y = point[1]  # Access: OK

point[0] = 15  # Error! Tuples are immutable
```

Tuples are perfect for data that shouldn't change, like coordinates, dates, or configuration values.

Sets: Unique, Unordered Collections

Sets store unique items with no duplicates. Order doesn't matter. Use curly braces `{}` or `set()` to create them.

```
unique_numbers = {1, 2, 3, 4, 5}
colors = {"red", "green", "blue"}

# Duplicates automatically removed
numbers = {1, 2, 2, 3, 3, 3}
print(numbers)  # {1, 2, 3}
```

Use cases: Removing duplicates, membership testing, mathematical set operations (union, intersection).

Set Operations

Add item

```
colors.add("yellow")
```

Remove item

```
colors.remove("red")
```

Union

```
set1 | set2
```

All items from both

Intersection

```
set1 & set2
```

Common items only

Difference

```
set1 - set2
```

Items in set1 but not set2

Dictionaries: Key-Value Pairs

Dictionaries store data as key-value pairs. Keys must be unique and immutable (strings, numbers, tuples). Values can be any type.

```
student = {
 "name": "Alice",
 "age": 20,
 "major": "Computer Science",
 "gpa": 3.8
}
```

Accessing Dictionary Values

Access values using keys, not numeric indices. Two ways to retrieve values:

```
# Using square brackets
print(student["name"])  # Alice

# Using get() method (safer)
print(student.get("age"))  # 20
print(student.get("city"))  # None (no error)
```

Why use get()?

If a key doesn't exist, `[]` raises an error, but `get()` returns `None` or a default value you specify.

```
student.get("city", "Unknown")
```

Modifying Dictionaries

Add/Update

```
student["city"] = "New York"
```

Creates key if it doesn't exist, updates if it does

Remove

```
del student["gpa"]
```

Deletes the key-value pair entirely

Pop

```
age = student.pop("age")
```

Removes key and returns its value

Get all keys

```
student.keys()
```

Returns list-like view of all keys

Get all values

```
student.values()
```

Returns list-like view of all values

Looping Through Dictionaries

You can iterate over keys, values, or both simultaneously.

```
# Loop through keys
for key in student:
    print(key)

# Loop through values
for value in student.values():
    print(value)

# Loop through both (most common)
for key, value in student.items():
    print(f"{key}: {value}")
```

The `.items()` method returns key-value pairs as tuples, which you can unpack in the loop.

Choosing the Right Data Structure

Lists

Need order? Items might repeat? Need to modify often? Use a list.

Example: Shopping cart items, student grades

Tuples

Data shouldn't change? Need to ensure integrity? Use a tuple.

Example: GPS coordinates, database records

Sets

Need unique items only? Order doesn't matter? Use a set.

Example: Email list (no duplicates), tags

Dictionaries

Need to look up values by name/ID? Key-value relationships? Use a dictionary.

Example: User profiles, configuration settings

REAL-WORLD EXAMPLE

Building a Simple Grade Tracker

Let's combine control flow and data structures to build something practical: a program that tracks student grades and calculates statistics.

Grade Tracker Code

```
# Store student grades in a dictionary
grades = {
 "Alice": [85, 90, 92],
 "Bob": [78, 82, 88],
 "Charlie": [95, 98, 100]
}

# Calculate and display averages
for student, scores in grades.items():
 total = sum(scores)
 average = total / len(scores)

 # Determine letter grade
 if average >= 90:
 letter = "A"
 elif average >= 80:
 letter = "B"
 else:
 letter = "C"

 print(f"{student}: {average:.1f} ({letter})")
```

PCAP EXAM TIPS

What to Remember for the Exam

1 **Indentation is syntax**

4 spaces define code blocks. Wrong indentation = errors.

2 **Know your operators**

Comparison (`==`, `!=`) vs assignment (`=`). Logical operators: `and`, `or`, `not`.

3 **Index starts at 0**

First item is `[0]`, last is `[-1]`. Remember slicing syntax.

4 **Mutability matters**

Lists are mutable, tuples and strings are not. Sets have no duplicates.

5 **Dictionary keys must be immutable**

Strings, numbers, tuples work. Lists and sets don't.

Practice Makes Perfect

Next Steps

- Code every concept yourself—don't just read
- Experiment with variations and edge cases
- Build small projects combining today's topics
- Review PCAP practice questions regularly

Key Takeaway

Control flow and data structures are the foundation of all programs. Master these, and you can build anything.

Remember: Understanding *why* to use each tool is just as important as knowing *how*.