PYTHON

Fundamentals

Getting Started · Variables & Data Types · Operators & Control Flow
Loops · Strings & Formatting · Lists, Tuples & Sets
Dictionaries · Functions · Modules & Packages

Philip Osztromok

Generated with Claude

Table of Contents

Python Fundamentals — 9 Chapters

Getting Started with Python

Python is a high-level, interpreted, general-purpose language prized for readability — it's used everywhere from quick scripts to web backends (Django and FastAPI, both already on this site) to data science and automation. This chapter covers what Python is and why it's worth learning, installing it, the interactive REPL, running scripts, `print()`, comments, and the `python3` vs `python` command distinction that trips up almost everyone at least once.

🌎 What Is Python, and Why Learn It?

Python is:

- **Interpreted** — no separate compile step; a script runs directly, unlike Go, Rust, or Kotlin, all already covered on this site
- **Dynamically typed** — a variable's type is determined at runtime, not checked at compile time (the opposite of Kotlin's or TypeScript's static typing)
- **General-purpose** — web backends, scripting/automation, data science, and more, all with the same core language
- **"Batteries included"** — a large standard library covers files, networking, dates, and more with no extra installs
- **Readability-focused** — indentation is part of the syntax itself, not just a style convention

Python vs Go vs Kotlin, at a Glance

Trait	Go	Kotlin	Python
Execution model	Compiled to a binary	Compiled to JVM bytecode	Interpreted directly
Typing	Static	Static, inferred	Dynamic
Block structure	Braces { }	Braces { }	Indentation only
Entry point	func main()	fun main()	No special entry function required

💾 Installing Python & Checking It Works

Install from python.org (Windows/Mac) or your system's package manager (`apt install python3` on Debian/Ubuntu, already familiar from the Linux courses). Verify the install from a terminal:

```
$ python3 --version
Python 3.12.1
```

▶️ The REPL — Python's Interactive Shell

Running `python3` with no arguments drops into an interactive shell — try expressions immediately, no file needed:

```
$ python3
Python 3.12.1 (main, Dec  7 2023, 00:00:00)
>>> 2 + 2
4
>>> print("hello")
hello
>>> exit()
```

Like Node's REPL

If you've used `node` with no arguments (Node.js Course 1), this is the exact same idea — an interactive prompt for trying things out immediately, no file or compile step needed.

Exiting the REPL

Type `exit()`, or press `Ctrl+D` (Mac/Linux) / `Ctrl+Z` then Enter (Windows).

📄 Running a Python Script

hello.py

```
# hello.py
print("Hello, Python!")
```

Run it:

```
$ python3 hello.py
Hello, Python!
```

No compile step, no entry-point function to declare — the script's top-level code just runs, top to bottom, the moment the interpreter reaches it.

🖨️ print() — Python's console.log

```
print("The answer is", 42)          # The answer is 42 — multiple args, auto-spaced
print("a", "b", "c", sep="-")       # a-b-c — sep changes the joiner
print("no newline", end="")         # end changes what's printed after (default: newline)
```

`print()` accepts any number of comma-separated arguments, joining them with a space and adding a trailing newline by default — both behaviors are customizable with the `sep` and `end` keyword arguments.

Comments

```
# A single-line comment
"""
A triple-quoted string used as a multi-line comment
or a docstring — common right under a function/class definition
"""
```

Python has no dedicated multi-line comment syntax like `/* */` — a triple-quoted string not assigned to anything is commonly used instead, and the same syntax doubles as a **docstring** when placed as the first line inside a function, class, or module (covered properly once functions are introduced in Chapter 8).

python3 vs python — Why It Matters

Python 2 reached end-of-life in 2020, but its legacy lives on in one confusing spot: on many systems, the bare `python` command still points at Python 2 (or doesn't exist at all), while `python3` reliably points at a modern Python 3 interpreter. Windows installers from python.org typically set up `python` to mean Python 3 directly, but Linux and Mac frequently don't.

⚠ Always Use python3 (and pip3) Explicitly

Running `python` on a system where it's aliased to Python 2 — or where it's missing entirely — is a common source of confusing "command not found" or bizarre syntax errors for beginners following an old tutorial. Get in the habit of typing `python3` and `pip3` explicitly from day one; it works everywhere, with zero ambiguity.

💻 Coding Challenges

Challenge 1: Your First Script

Create a file called `about_me.py` that prints your name and a short greeting, using two separate `print()` calls. Run it with `python3 about_me.py`.

Goal: Get comfortable creating and running a Python script from the command line.

→ Solution

Challenge 2: print() Options

Write a single `print()` call that prints three words separated by `" | "` instead of a space, using the `sep` keyword argument.

Goal: Practice using keyword arguments to customize a built-in function's behavior.

→ Solution

Challenge 3: REPL Exploration

Open the REPL with `python3` and try five different expressions of your choice (arithmetic, string concatenation with `+`, etc.). Write down what each one returned.

Goal: Build comfort using the REPL as a quick scratchpad, not just for running scripts.

→ Solution

💡 IDLE: Python's Bundled Editor

Python's official installer includes **IDLE**, a simple built-in editor with its own REPL window — genuinely fine for these early chapters. This course doesn't require any particular editor; use IDLE, VS Code, or whatever you're already comfortable with.

🎯 What's Next

Next chapter: **Variables & Basic Data Types** — `int`/`float`/`str`/`bool`, Python's dynamic typing in practice, and using `type()` to check and convert between types.

🔢 Variables & Basic Data Types

Chapter 1 got scripts and the REPL running. This chapter covers how Python actually stores values — no declaration keyword at all, dynamic typing that lets a variable hold any type at any time, the four basic data types, and converting between them with `type()` and casting.

📝 Declaring Variables — No Keyword Needed

Python has no `let`, `var`, or `val` — just a name, `=`, and a value:

```
name = "Ada"
age = 30
```

Declaring a Value: Go vs Kotlin vs Python

Language	Syntax
Go	age := 30 / var age int = 30
Kotlin	val age = 30 / var age = 30
Python	age = 30

🔄 Dynamic Typing: Types Belong to Values, Not Variables

This is a genuinely meaningful difference from Go and Kotlin, both already covered — in Python, a variable can be reassigned to a **completely different type** at any point:

```
x = 5 # x is currently an int
print(type(x))   # <class 'int'>

x = "hello" # perfectly valid — x is now a str
print(type(x))   # <class 'str'>
```

In Kotlin, `val age = 30` locks `age` to `Int` forever — assigning a `String` to it later is a compile error. In Python, the *type* belongs to the value currently stored, not to the variable name itself; the same name can point at an `int` one moment and a `str` the next.

⚠ Convenient, but a Real Source of Subtle Bugs

Dynamic typing means a typo or a logic mistake that assigns the wrong type to a variable often won't be caught until the program actually runs and fails — unlike Go or Kotlin, where the compiler catches this before the program ever executes. Python's answer to this trade-off is **type hints** (Course 3's own dedicated chapter) — an opt-in way to declare intended types that tools like `mypy` can check, without changing Python's actual runtime behavior.

🧱 The Basic Data Types

```
age = 30 # int — whole numbers
price = 19.99 # float — decimal numbers
name = "Philip" # str — text
is_ready = True # bool — True or False (capitalized!)
```

int

Whole numbers, positive or negative, no fixed size limit (unlike Go's `int32`/`int64`).

float

Decimal numbers. Any number written with a decimal point is automatically a `float`.

str

Text, in single or double quotes — Python treats `'text'` and `"text"` identically.

bool

`True` or `False` — capitalized, unlike Go/Kotlin's lowercase `true`/`false`.

🔍 type() — Checking a Value's Type

```
print(type(30))       # <class 'int'>
print(type(19.99))    # <class 'float'>
print(type("hi"))     # <class 'str'>
print(type(True))     # <class 'bool'>
```

🔁 Type Casting — Converting Between Types

```
int("42")      # 42 — str to int
str(42)        # "42" — int to str
float("3.14")  # 3.14 — str to float
int(3.99)      # 3 — float to int (truncates, doesn't round!)
```

⚠ int("3.14") Fails — A Genuine Common Mistake

`int("3.14")` raises a `ValueError`, not `3` — `int()` can't parse a decimal point directly from a string. The fix is going through `float` first: `int(float("3.14"))` → `3`. This trips up nearly everyone the first time they try to convert user input that happens to contain a decimal.

✅ Truthiness — What Counts as False

Every value in Python has an implicit boolean meaning, checked with `bool()`: `0`, `0.0`, `""` (empty string), `None`, and empty collections are all **falsy**; everything else is **truthy**. This becomes directly useful once `if` statements arrive in the next chapter.

Static Typing (Go/Kotlin) vs Dynamic Typing (Python)

	Go / Kotlin	Python
Type belongs to	The variable, fixed at declaration	The current value, can change
Type mismatch caught	At compile time	At runtime (if at all)
Type hints	Required (Go) / default (Kotlin)	Optional, opt-in (Course 3)

💻 Coding Challenges

Challenge 1: Four Types, One Script

Declare one variable of each basic type (`int`, `float`, `str`, `bool`), then print each one alongside its type using `type()`.

Goal: Get comfortable with all four basic types and confirming them with `type()`.

→ Solution

Challenge 2: The int("3.14") Trap

Write code that safely converts the string `"3.99"` into an integer, working around the fact that `int()` can't parse a decimal point directly.

Goal: Practice the float-then-int casting workaround from this chapter's warn-box.

→ Solution

Challenge 3: Dynamic Typing in Action

Write code that assigns an `int` to a variable, prints its type, then reassigns the same variable name to a `str`, and prints its type again — proving the type genuinely changed.

Goal: Build a concrete, working example of dynamic typing rather than just reading about it.

→ Solution

🎯 What's Next

Next chapter: **Operators & Control Flow** — arithmetic/comparison/logical operators, `if`/`elif`/`else`, and the walrus operator.

⚙️ Operators & Control Flow

Chapter 2 covered Python's basic types. This chapter covers combining and comparing values with operators, and branching logic with `if/elif/else` — plus a genuinely modern addition, the walrus operator.

➕ Arithmetic Operators

```
7 + 3 # 10
7 - 3 # 4
7 * 3 # 21
7 / 3 # 2.3333333333333335 — always a float (Chapter 2)
7 // 3 # 2 — floor division, discards the remainder
7 % 3 # 1 — modulo, the remainder
7 ** 2 # 49 — exponentiation, built directly into the operator
```

Exponentiation: Go/Kotlin vs Python

Neither Go nor Kotlin has an exponent operator at all — both require a function call (`math.Pow(7, 2)` in Go, `Math.pow(7.0, 2.0)` in Kotlin). Python's `**` is a genuine convenience built directly into the language's operator set.

⚖️ Comparison Operators

```
5 == 5 # True
5 != 3 # True
5 > 3 # True
5 <= 3 # False
```

`==` compares by **value** — `"abc" == "abc"` is `True`, even though they might be two separately-created strings. A separate operator, `is`, checks whether two names point at the exact same object in memory — a distinction that matters more once mutable objects like lists arrive in Chapter 6.

🔗 Logical Operators

```
True and False # False
True or False # True
not True # False
```

Logical Operators: Symbols vs Words

	Go / Kotlin	Python
AND	&&	and
OR	||	or
NOT	!	not

Python spells logical operators out as words instead of symbols — a small but real readability choice consistent with the language's overall design philosophy.

🔀 if / elif / else

```
age = 20
if age < 13:
    print("child")
elif age < 20:
    print("teenager")
else:
    print("adult")   # adult
```

No braces at all — the colon `:` starts a block, and consistent indentation defines what's inside it, exactly the "significant whitespace" Chapter 1 mentioned. `elif` is Python's spelling of "else if," used in every conditional chain instead of Go/Kotlin's `else if`.

⚠ Indentation Errors Are a Real, Common Beginner Trap

Mixing tabs and spaces, or indenting one line of a block slightly differently than its siblings, produces an `IndentationError` — Python is strict about this since indentation isn't just style, it's the actual block structure. Most editors (including IDLE) convert Tab to 4 spaces automatically to avoid this; stick to spaces consistently.

The Walrus Operator (:=)

Added in Python 3.8, the walrus operator assigns a value **and** uses it in the same expression — useful for avoiding a repeated function call:

```
data = [1, 2, 3, 4, 5]

if (n := len(data)) > 3:
    print(f"list is too long: {n} items")   # list is too long: 5 items
```

Without the walrus operator, this would need `len(data)` called twice — once inside the condition, once again inside the `print` — or a separate line assigning `n` before the `if`. `:=` does both in one expression.

Arithmetic

+ - * / // % ** — including a built-in exponent operator, unlike Go or Kotlin.

Comparison

== != < > <= >= — value equality by default; `is` checks identity.

Logical

`and` / `or` / `not` — spelled as words, not symbols.

Walrus (:=)

Assign and use a value in one expression — genuinely new syntax, added in 3.8.

💻 Coding Challenges

Challenge 1: FizzBuzz-Style Check

Write an `if`/`elif`/`else` chain that checks a number: if it's divisible by both 3 and 5, print "FizzBuzz"; if only by 3, print "Fizz"; if only by 5, print "Buzz"; otherwise print the number itself.

Goal: Combine comparison operators, the modulo operator, and logical `and` in one realistic conditional chain.

→ Solution

Challenge 2: Floor Division vs Modulo

Given a total number of minutes, write code using `//` and `%` to compute and print the equivalent hours and remaining minutes (e.g. 125 minutes → 2 hours, 5 minutes).

Goal: Practice using floor division and modulo together for a genuinely practical calculation.

→ Solution

Challenge 3: Walrus in a Loop-Free Context

Given a string, use the walrus operator inside an `if` condition to check whether its length exceeds 10, printing the actual length only if the condition is true — without calling `len()` twice.

Goal: Get hands-on with `:=` in the simplest realistic case before it appears again in later chapters.

→ Solution

🎯 What's Next

Next chapter: **Loops** — `for` and `while`, `range()`, `break`/`continue`/the loop `else` clause, and `enumerate()`.

🔁 Loops

Chapter 3 covered branching. This chapter covers repetition — `for` and `while` loops, `range()`, loop control with `break`/`continue`, a genuinely unusual Python feature (the loop `else` clause), and `enumerate()`.

🔁 for Loops: Iterating Over a Sequence

Python's `for` iterates directly over items in a sequence — not an index counter, unlike a C-style `for`:

```
for char in "abc":
    print(char)
# a
# b
# c
```

Looping: Go vs Kotlin vs Python

Go uses a single `for` keyword for every loop shape (counting, condition-based, infinite) — there's no separate `while` at all. Kotlin's `for (item in collection)` is directly comparable to Python's item-based `for`. Python keeps `for` (iterate over items) and `while` (loop on a condition) as two distinct keywords.

🔢 range(): Looping a Specific Number of Times

```
for i in range(5):      # 0, 1, 2, 3, 4 — stop is EXCLUSIVE
print(i)

for i in range(1, 5):   # 1, 2, 3, 4 — start, stop
print(i)

for i in range(0, 10, 2):  # 0, 2, 4, 6, 8 — start, stop, step
print(i)
```

⚠ range(5) Stops at 4, Not 5

`range(n)`'s `stop` value is always **exclusive** — `range(5)` produces `0, 1, 2, 3, 4`, five numbers total, never `5` itself. This trips up nearly everyone the first time they need exactly `n` iterations and reach for `range(n)` expecting it to count up to `n` inclusive.

while Loops

```
count = 0
while count < 3:
    print(count)
    count += 1
# 0
# 1
# 2
```

Python has no `do-while` construct at all — the condition is always checked before the loop body runs, at least once via `while` or not at all.

⏭️ break and continue

```
for n in range(10):
    if n == 5:
        break # stop the loop entirely
if n % 2 == 0:
        continue # skip to the next iteration
print(n)            # 1, 3
```

🎯 The Loop else Clause

A genuinely distinctive Python feature: `for` and `while` loops can have an `else` clause that runs only if the loop finishes **without** hitting a `break` — useful for search patterns:

```
numbers = [2, 4, 6, 8]

for n in numbers:
    if n % 2 != 0:
        print("found an odd number")
        break
else:
    print("all numbers were even")   # this runs — no break occurred
```

Neither Go nor Kotlin has anything like this — it's a Python-specific idiom worth recognizing when reading other people's code, even if you don't reach for it constantly yourself.

🔢 enumerate(): Getting the Index While Iterating

```
for i, char in enumerate("abc"):
    print(i, char)
# 0 a
# 1 b
# 2 c
```

`enumerate()` pairs each item with its index automatically — no separate counter variable to manually increment, the way a C-style loop would need.

for

Iterates directly over items in a sequence — strings, lists, `range()`, and more.

while

Loops on a condition, checked before each iteration; no do-while equivalent exists.

break / continue

Exit the loop entirely, or skip straight to the next iteration.

enumerate()

Pairs each item with its index — no manual counter variable needed.

💻 Coding Challenges

Challenge 1: Sum of Even Numbers

Using `range()` and a `for` loop, calculate and print the sum of all even numbers from 1 to 20 (inclusive).

Goal: Practice combining `range()`, the modulo operator, and loop accumulation.

→ Solution

Challenge 2: Search With Loop else

Given a list of names, write a `for` loop with a matching `else` clause that prints "found!" and breaks if a target name is in the list, or prints "not found" via the `else` clause if it never is.

Goal: Get hands-on with the loop `else` clause in the exact search scenario it's designed for.

→ Solution

Challenge 3: Numbered List Output

Given a list of three favorite foods, use `enumerate()` to print each one with a 1-based number in front of it (e.g. "1. Pizza"), not the default 0-based index.

Goal: Practice using `enumerate()` and adjusting its default starting index for a human-friendly numbered list.

→ Solution

🎯 What's Next

Next chapter: **Strings & String Formatting** — slicing, string methods, f-strings, and `.format()`.

🔤 Strings & String Formatting

You've been using strings since Chapter 1 without a close look at what they can do. This chapter covers indexing and slicing, string immutability, the most common string methods, and Python's three ways to build formatted strings — f-strings, `.format()`, and old-style `%` — with a clear recommendation for which one to actually reach for.

✂️ String Indexing & Slicing

```
word = "Python"
print(word[0])     # P — first character
print(word[-1])    # n — last character, negative index
print(word[1:4])   # yth — slice: start inclusive, stop exclusive
print(word[:3])    # Pyt — omit start, defaults to 0
print(word[3:])    # hon — omit stop, defaults to the end
print(word[::-1])  # nohtyP — step of -1 reverses the string
```

Slice syntax is `[start:stop:step]` — the same shape you'll see again on lists in Chapter 6. `stop` is exclusive, matching `range()` from the last chapter.

String Indexing: Go vs Kotlin vs Python

Language	Indexing a string
Go	A string is a byte slice — indexing with `s[0]` gives a raw byte, not a character. Multi-byte UTF-8 characters (like emoji or accented letters) require iterating with `range` to get correct character boundaries.
Kotlin	`s[0]` gives a `Char` — indexing is character-based like Python, inherited from Java's `String` behavior.
Python	`s[0]` gives a single-character string, always Unicode-aware — no separate "byte vs character" distinction to worry about at this level.

🔒 Strings Are Immutable

A string can't be changed in place — every string method that looks like it's "modifying" a string is actually returning a brand-new string:

```
name = "python"
upper_name = name.upper()

print(name)       # python — unchanged
print(upper_name)  # PYTHON — a new string
```

🛠️ Common String Methods

```
s = "  Hello, World  "

s.strip()              # "Hello, World" — removes leading/trailing whitespace
s.lower()              # "  hello, world  "
s.upper()              # "  HELLO, WORLD  "
s.strip().split(", ")  # ['Hello', 'World']
"-".join(["a", "b", "c"])  # "a-b-c"
s.replace("World", "Python")  # "  Hello, Python  "
s.strip().startswith("Hello")  # True
s.strip().endswith("World")    # True
s.find("World")          # 9 — index where it starts, or -1 if not found
```

💡 join() Lives on the Separator, Not the List

`"-".join(["a", "b", "c"])` reads backwards to most newcomers — the separator string comes first, and the list of pieces to join is the argument. Building a string by repeatedly concatenating with `+` inside a loop works, but `"".join(...)` is the idiomatic and more efficient way to combine many pieces at once, since strings are immutable and each `+` creates yet another new string.

📝 f-strings: Formatted String Literals

An f-string embeds expressions directly inside a string, prefixed with `f`:

```
name = "Ada"
age = 28
print(f"{name} is {age} years old")
# Ada is 28 years old
print(f"Next year: {age + 1}")    # expressions work directly inside {}
# Next year: 29

pi = 3.14159265
print(f"Pi rounded: {pi:.2f}")   # format spec after the colon
# Pi rounded: 3.14
```

📐 .format() and % — The Older Styles

```
# .format() — pre-f-string standard, still common in older code
print("{} is {} years old".format(name, age))

# % formatting — the oldest style, inherited from C's printf
print("%s is %d years old" % (name, age))
```

Which Style Should You Use?

f-strings (added in Python 3.6) are the modern, idiomatic choice — they're more readable since the variable sits directly inside the string instead of matched up positionally, and they're generally faster. `.format()` and `%` both still appear constantly in real-world and older code, so recognizing them matters even though you should default to f-strings in new code you write.

Slicing

`s[start:stop:step]` — stop is exclusive, negative indices count from the end.

Immutability

String methods never modify in place — they always return a new string.

String Methods

`.strip()`, `.split()`, `.join()`, `.replace()`, `.find()` and more.

f-strings

`f"{expr:.2f}"` — the modern, idiomatic way to build formatted strings.

💻 Coding Challenges

Challenge 1: Reverse Words

Given the string `"Python is fun"`, use `.split()` and `.join()` to print the words in reverse order: `"fun is Python"`.

Goal: Practice combining `.split()` and `.join()` with slicing to reverse a sequence.

→ Solution

Challenge 2: Formatted Receipt Line

Given `item = "Coffee"`, `price = 4.5`, and `qty = 3`, use an f-string to print a line like `"3x Coffee: $13.50"`, with the total formatted to exactly 2 decimal places.

Goal: Practice f-string expressions and the `:.2f` format spec together.

→ Solution

Challenge 3: Palindrome Check

Write code that checks whether the string `"racecar"` is a palindrome (reads the same forwards and backwards), using slicing rather than a loop.

Goal: Practice using the `[::-1]` slice pattern to solve a real small problem.

→ Solution

🎯 What's Next

Next chapter: **Lists, Tuples & Sets** — mutable vs. immutable sequences, and a first look at comprehensions.

📦 Lists, Tuples & Sets

Python has three built-in collection types worth knowing apart from day one: **lists** (mutable, ordered), **tuples** (immutable, ordered), and **sets** (unordered, unique). This chapter covers all three, plus a first look at list comprehensions — a compact way to build a list from an existing sequence.

📋 Lists — Mutable, Ordered Sequences

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

fruits.append("date")      # ['apple', 'banana', 'cherry', 'date']
fruits.insert(1, "apricot") # insert at a specific index
fruits.remove("banana")    # removes the first matching value
last = fruits.pop()        # removes and returns the last item
fruits.sort()              # sorts in place, alphabetically
print(fruits[0])      # indexing and slicing work exactly like strings
print(fruits[1:])
```

Lists use the same indexing and slicing syntax you already know from strings (Chapter 5) — `[start:stop:step]`, negative indices, all of it.

⚠ Assigning a List Copies the Reference, Not the List

```
a = [1, 2, 3]
b = a        # b now points to the SAME list as a
b.append(4)
print(a)     # [1, 2, 3, 4] — a changed too!
```

`b = a` does not create a second, independent list — it makes `b` a second name for the exact same list object in memory, since lists are mutable. Modifying `b` modifies the only list that exists. To get a real independent copy, use `b = a.copy()` or `b = a[:]`.

🔒 Tuples — Immutable, Ordered Sequences

```
point = (3, 7)
print(point[0])  # 3 — indexing works like a list
# point[0] = 5  # TypeError — tuples can't be modified after creation
# Tuple unpacking — a very common Python idiom
x, y = point
print(x, y)  # 3 7
```

Tuples exist for data that shouldn't change after creation — coordinates, RGB values, a fixed record returned from a function. Because they're immutable, they can also be used as dictionary keys and set members (Chapter 7 covers dictionaries), which lists cannot.

🎯 Sets — Unordered, Unique Collections

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

a = {1, 2, 3}
b = {2, 3, 4}

print(a | b)   # {1, 2, 3, 4} — union
print(a & b)   # {2, 3} — intersection
print(a - b)   # {1} — difference
print(3 in a)   # True — membership tests on sets are very fast
```

Collections: Go vs Kotlin vs Python

Concept	Go	Kotlin	Python
Mutable ordered	`[]T` (slice)	`MutableList<T>`	list
Fixed-size ordered	`[N]T` (array)	`List<T>` (read-only view)	tuple
Unique/unordered	no built-in set — usually `map[T]bool`	`MutableSet<T>`	set

Go's lack of a built-in set type is a real, sometimes-surprising gap coming from Python — reaching for `map[T]bool` as a workaround is the standard idiom there.

⚡ A First Look at List Comprehensions

A list comprehension builds a new list from an existing sequence in a single, compact expression:

```
numbers = [1, 2, 3, 4, 5]

# The loop version, from Chapter 4:
squares = []
for n in numbers:
    squares.append(n ** 2)

# The comprehension version — same result, one line:
squares = [n ** 2 for n in numbers]

# With a filtering condition:
even_squares = [n ** 2 for n in numbers if n % 2 == 0]
print(even_squares)  # [4, 16]
```

💡 Just a First Look

This is a preview, not the full picture — comprehensions get their own dedicated deep dive in Course 2 (dict/set comprehensions, nested comprehensions, and generator expressions). For now, just recognize the pattern: `[expression for item in iterable if condition]`.

list

Mutable, ordered — the everyday, general-purpose collection.

tuple

Immutable, ordered — for fixed data, unpacking, and use as dict keys.

set

Unordered, unique — automatic deduplication and fast membership tests.

List comprehension

`[expr for item in iterable if condition]` — a compact way to build a list.

💻 Coding Challenges

Challenge 1: Remove Duplicates, Keep Order

Given `numbers = [3, 1, 3, 2, 1, 4, 2]`, produce a new list with duplicates removed but the original first-seen order preserved: `[3, 1, 2, 4]`. (Hint: a plain `set()` alone won't preserve order.)

Goal: Understand the real trade-off between sets (fast, unordered) and lists (ordered) firsthand.

→ Solution

Challenge 2: Coordinate Unpacking

Given a list of three tuples, `points = [(1, 2), (3, 4), (5, 6)]`, loop over it and print each point as `"x=1, y=2"` using tuple unpacking directly in the `for` loop header.

Goal: Practice tuple unpacking combined with iteration.

→ Solution

Challenge 3: Common Elements

Given two lists, `a = [1, 2, 3, 4, 5]` and `b = [3, 4, 5, 6, 7]`, use sets to find and print the elements that appear in both, as a sorted list: `[3, 4, 5]`.

Goal: Practice converting lists to sets to use set intersection, then converting back.

→ Solution

🎯 What's Next

Next chapter: **Dictionaries** — key-value operations, iterating, and dict comprehensions.

📖 Dictionaries

A dictionary stores key-value pairs — the tool you reach for whenever you need to look something up by a name rather than a position. This chapter covers creating and accessing dictionaries, modifying them, the three ways to iterate over one, and a first dict comprehension.

🔑 Creating and Accessing Dictionaries

```
person = {
    "name": "Ada",
    "age": 28,
    "language": "Python"
}

print(person["name"])      # Ada
print(person.get("age"))    # 28
print(person.get("email", "unknown"))  # unknown — default if key is missing
```

⚠ Direct Indexing Raises KeyError on a Missing Key

```
person["email"]  # KeyError: 'email' — crashes the program
```

`person["email"]` raises a `KeyError` immediately if the key doesn't exist — there's no silent `None` the way some languages return. Use `.get()` whenever a key might legitimately be missing; it returns `None` (or your own default value) instead of crashing.

📝 Modifying Dictionaries

```
person["email"] = "ada@example.com" # adds a new key
person["age"] = 29 # overwrites an existing key
del person["language"]            # removes a key entirely
email = person.pop("email")       # removes AND returns the value
print("age" in person)         # True — membership testing checks KEYS
```

🔁 Iterating Over Dictionaries

```
scores = {"Alice": 90, "Bob": 85, "Charlie": 78}

for name in scores.keys():
    print(name)                  # Alice, Bob, Charlie
for score in scores.values():
    print(score)                 # 90, 85, 78
for name, score in scores.items():
    print(f"{name}: {score}")    # Alice: 90, Bob: 85, Charlie: 78
```

Plain `for name in scores:` (no `.keys()`) also works and iterates over keys — `.keys()` is optional but makes the intent explicit. `.items()` is the one to reach for whenever you need both the key and value together, pairing naturally with the tuple unpacking from the previous chapter.

Key-Value Collections: Go vs Kotlin vs Python

Language	Type	Missing key behavior
Go	`map[string]int`	Returns the zero value silently (e.g. `0`) — no crash, no error, which can hide real bugs. The comma-ok idiom (`v, ok := m[key]`) is needed to actually detect a missing key.
Kotlin	`Map<K, V>` / `MutableMap<K, V>`	`map[key]` returns a nullable type (`V?`); `map.getValue(key)` throws if missing, directly comparable to Python's bracket-indexing/`.get()` split.
Python	dict	`d[key]` raises `KeyError` immediately; `d.get(key, default)` returns a default instead.

Dict Comprehensions

The same comprehension pattern from the last chapter works for dictionaries too, using `{key: value for ...}`:

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

squares = {n: n ** 2 for n in numbers}
print(squares)  # {1: 1, 2: 4, 3: 9, 4: 16}
# with a filtering condition:
even_squares = {n: n ** 2 for n in numbers if n % 2 == 0}
print(even_squares)  # {2: 4, 4: 16}
```

Creating & Accessing

`{key: value}` literal, `d[key]` indexing, `d.get(key, default)` for safety.

Modifying

Assign to add/overwrite, `del d[key]` or `d.pop(key)` to remove.

Iterating

`.keys()`, `.values()`, `.items()` — `.items()` for key+value pairs together.

Dict comprehension

`{k: v for k, v in ... if condition}` — same pattern as list comprehensions.

💻 Coding Challenges

Challenge 1: Word Frequency Counter

Given the list `words = ["apple", "banana", "apple", "cherry", "banana", "apple"]`, build a dictionary counting how many times each word appears, using a loop and `.get()` with a default.

Goal: Practice the common "build a count dictionary from a list" pattern.

→ Solution

Challenge 2: Invert a Dictionary

Given `capitals = {"France": "Paris", "Japan": "Tokyo", "Italy": "Rome"}`, build a new dictionary with the keys and values swapped — `{"Paris": "France", "Tokyo": "Japan", "Rome": "Italy"}` — using a dict comprehension and `.items()`.

Goal: Combine `.items()` iteration with a dict comprehension in one line.

→ Solution

Challenge 3: Safe Lookup Report

Given `inventory = {"apples": 10, "bananas": 5}` and a list of items to check, `["apples", "bananas", "cherries"]`, print a stock report line for each one — using `.get()` so a missing item like `"cherries"` prints `"0 in stock"` instead of crashing.

Goal: Practice defensive dictionary lookups with `.get()` instead of direct indexing.

→ Solution

🎯 What's Next

Next chapter: **Functions** — `def`, positional/keyword/default/`*args`/`**kwargs` parameters, `return`, and scope.

🧩 Functions

Everything so far has been one script running top to bottom. Functions let you package up reusable pieces of logic — this chapter covers defining and calling them, the four ways to pass parameters (positional, keyword, default, and variadic), `return`, and how variable scope actually works inside a function body.

🛠️ Defining and Calling Functions

```
def greet(name):
    return f"Hello, {name}!"

message = greet("Ada")
print(message)   # Hello, Ada!
```

Like everything else in Python, the function body is defined by indentation, not braces — the same rule that's applied to every `if`, `for`, and `while` block since Chapter 3.

📥 Parameters: Positional, Keyword, and Default

```
def describe_pet(name, animal="dog"):   # animal has a default value
return f"{name} is a {animal}"
print(describe_pet("Rex"))              # Rex is a dog — positional, uses default
print(describe_pet("Whiskers", "cat"))  # Whiskers is a cat — positional
print(describe_pet(name="Rex", animal="cat"))  # keyword arguments — order doesn't matter
print(describe_pet(animal="parrot", name="Polly"))
```

Calling with `name=...` instead of relying on position is a **keyword argument** — it makes the call self-documenting and lets you skip default parameters you don't need to override, or reorder arguments freely.

📦 *args and **kwargs

Sometimes you don't know in advance how many arguments a function will receive. `*args` collects extra positional arguments into a tuple; `**kwargs` collects extra keyword arguments into a dictionary:

```
def total(*args):
    return sum(args)

print(total(1, 2, 3, 4))  # 10 — args is (1, 2, 3, 4)
def build_profile(**kwargs):
    return kwargs

print(build_profile(name="Ada", age=28))
# {'name': 'Ada', 'age': 28} — kwargs is a dict
```

The names `args` and `kwargs` are convention, not a language rule — `*` and `**` are what actually matter. You'll see `*args, **kwargs` constantly in real Python code, especially in functions designed to wrap or forward calls to other functions.

↩️ Return Values

```
def no_return():
    print("did something")
    # no return statement

result = no_return()
print(result)   # None — a function with no return implicitly returns None
def min_max(numbers):
    return min(numbers), max(numbers)   # actually returns a tuple

low, high = min_max([3, 7, 1, 9])   # unpacked, same as Chapter 6
print(low, high)   # 1 9
```

"Returning multiple values" is really just returning a single tuple that gets unpacked at the call site — the same tuple mechanics from Chapter 6, not a separate language feature.

🔍 Scope

```
count = 10 # a global variable
def increment():
    count = 20 # this creates a NEW local variable, doesn't touch the global one
print(count)   # 20

increment()
print(count)   # 10 — unchanged
def increment_global():
    global count
    count += 1 # now this DOES modify the global

increment_global()
print(count)   # 11
```

⚠ Mutable Default Arguments Are a Classic Trap

```
def add_item(item, basket=[]):   # DANGER: default list created ONCE, not per-call
    basket.append(item)
    return basket

print(add_item("apple"))    # ['apple']
print(add_item("banana"))   # ['apple', 'banana'] — NOT a fresh empty list!
```

A default value like `basket=[]` is evaluated exactly once, when the function is defined — not fresh on every call. Since lists are mutable (Chapter 6), every call that relies on the default ends up sharing and mutating the *same* list. The standard fix is `basket=None`, then `if basket is None: basket = []` inside the function body.

Function Parameters: Go vs Kotlin vs Python

Feature	Go	Kotlin	Python
Default parameter values	Not supported — simulate with variadic params or a config struct	Supported directly, same idea as Python	Supported directly
Keyword/named arguments	Not supported	Supported directly (named arguments)	Supported directly
Variadic parameters	`func f(nums ...int)`	`fun f(vararg nums: Int)`	`def f(*args)`

Kotlin's default and named parameters map almost one-to-one onto Python's — this is one of the friendlier corners of the language to pick up coming from Kotlin. Go's lack of both is a real, common source of friction going the other direction.

def / return

Defines a function; a missing `return` implicitly returns `None`.

Default & keyword args

`def f(x, y=default)`; call with `f(y=1, x=2)` in any order.

*args / **kwargs

Collect extra positional args into a tuple, extra keyword args into a dict.

Scope

Assigning inside a function creates a local variable by default; `global` opts out of that.

💻 Coding Challenges

Challenge 1: Flexible Greeting

Write a function `greet(name, greeting="Hello")` that returns `f"{greeting}, {name}!"`. Call it three ways: with just a name, with both arguments positionally, and with both arguments as keyword arguments in reversed order.

Goal: Get comfortable with default values and keyword-argument calling in practice.

→ Solution

Challenge 2: Variadic Average

Write a function `average(*args)` that returns the average of however many numbers it's called with. Call it with 2 numbers, then again with 5 numbers.

Goal: Practice writing and calling a function with `*args`.

→ Solution

Challenge 3: Fix the Mutable Default Bug

Given the buggy `add_item(item, basket=[])` function from this chapter's warning box, rewrite it correctly using the `basket=None` pattern so that each call without a `basket` argument gets its own fresh, independent list.

Goal: Apply the fix for the mutable-default-argument gotcha directly, not just recognize it.

→ Solution

🎯 What's Next

Next chapter: **Modules & Packages** — `import`, pip, and virtual environments (venv).

📦 Modules & Packages

The final chapter of Python Fundamentals: how to use code beyond a single script — importing modules from the standard library, installing third-party packages with `pip`, and keeping each project's dependencies isolated with a virtual environment.

📥 Importing Modules

```
import math
print(math.sqrt(16))      # 4.0 — access via math.name
import math as m
print(m.pi)               # 3.14159... — a shorter alias
from math import sqrt, pi
print(sqrt(16))           # 4.0 — imported names used directly, no math. prefix
from math import *        # imports everything — see the warning below
```

⚠ Avoid from module import *

The star-import form dumps every public name from a module directly into your current namespace. It's convenient in a quick throwaway script, but in real code it makes it unclear where a given name actually came from, and it can silently overwrite (shadow) names you've already defined — including built-ins. Stick to `import module` or `from module import specific_name` instead.

📚 The Standard Library

Python ships with a large "batteries included" standard library — modules available immediately with no installation step:

```
import random
print(random.randint(1, 10))   # a random integer from 1 to 10, inclusive
import datetime
print(datetime.date.today())   # today's date
import os
print(os.getcwd())             # the current working directory
```

📦 pip and Third-Party Packages

`pip` is Python's package installer, used for code outside the standard library:

```
# at the command line, not inside a .py file:
pip install requests

# then, in your script:
import requests
```

A project's dependencies are typically tracked in a `requirements.txt` file — one package (and optionally a version) per line — so anyone else can recreate the exact same set with `pip install -r requirements.txt`.

🧪 Virtual Environments (venv)

Installing packages globally on your machine causes real problems the moment two projects need different, conflicting versions of the same package. A **virtual environment** gives each project its own isolated set of installed packages:

```
# create a virtual environment (once, per project)
python -m venv venv

# activate it — macOS/Linux
source venv/bin/activate

# activate it — Windows
venv\Scripts\activate

# now pip install only affects THIS project's environment
pip install requests

# leave the virtual environment
deactivate
```

💡 Always Use a Virtual Environment

Creating a fresh `venv` for every new Python project is standard practice, not an optional extra — it's the direct equivalent of Node's per-project `node_modules` folder, keeping each project's dependencies self-contained and reproducible on another machine.

Dependency Management: Go vs Kotlin vs Python

Language	Approach
Go	`go.mod`/`go.sum` declare and lock dependency versions per-project automatically — isolation is built into the tooling, no separate "activate an environment" step.
Kotlin	Gradle (or Maven) resolves dependencies declared in `build.gradle.kts`, scoped per-project by default — also no manual activation step.
Python	`pip` installs packages, but isolation isn't automatic — a `venv` has to be deliberately created and activated per project, or packages land in one shared global location.

This is a genuine rough edge coming from Go or Kotlin — both of those ecosystems bake project-level isolation into the build tool itself, while Python's isolation is a separate, opt-in step you have to remember to take.

import

`import x`, `import x as y`, `from x import y` — avoid `from x import *`.

Standard library

`math`, `random`, `datetime`, `os`, and many more — no install needed.

pip

Installs third-party packages; dependencies tracked in `requirements.txt`.

venv

An isolated, per-project set of installed packages — create and activate before installing anything.

💻 Coding Challenges

Challenge 1: Random Dice Roller

Using the `random` module, write a function `roll_dice()` that returns a random integer between 1 and 6 (inclusive), then call it and print the result.

Goal: Practice importing and using a standard library module.

→ Solution

Challenge 2: Days Until a Date

Using the `datetime` module, calculate and print how many days remain until January 1st of next year, starting from today's date.

Goal: Practice working with the `datetime` module and date arithmetic.

→ Solution

Challenge 3: Write a requirements.txt

Write out (as plain text, no code needed to run) a `requirements.txt` file listing three made-up third-party packages your project depends on, each pinned to a specific version using the `==` syntax.

Goal: Get familiar with the `requirements.txt` format itself, since you'll be reading and writing these in almost every real Python project.

→ Solution

🎓 Course Complete!

That's all 9 chapters of **Python Fundamentals** — installation, variables, control flow, loops, strings, lists/tuples/sets, dictionaries, functions, and modules/packages. Next up: **Python Intermediate** (`py2`), starting with Object-Oriented Programming.