PYTHON
ZERO TO HERO

A complete beginner-to-professional Python course — readable in one sitting.

Covers syntax, data structures, functions, error handling, object-oriented programming, and the critical mistakes that trip up every beginner.

20 CHAPTERS • ZERO PRIOR EXPERIENCE REQUIRED

Contents

How to use this course: read top to bottom, once. Every chapter builds on the last. Boxes marked ■ **CRITICAL** flag mistakes that even experienced developers make — slow down and read those twice.

CHAPTER 01

What Is Python & How It Runs

Python is an **interpreted, dynamically-typed** programming language. "Interpreted" means your code runs line by line through a program called the Python interpreter — there is no separate compile step you have to manage. "Dynamically-typed" means you don't declare the type of a variable; Python figures it out at runtime. This makes Python fast to write and easy to read, at the cost of some raw execution speed compared to compiled languages like C or Rust.

Installing Python

Download Python 3 from python.org (use Python 3, not Python 2 — Python 2 stopped receiving updates in 2020). On macOS and Linux it may already be installed; check with the command below.

```
python3 --version
# python3 is preferred; on some systems 'python' still points to v2
```

Running Your First Program

Create a file named `hello.py` with one line, then run it from a terminal.

```
print("Hello, World!")
```

hello.py

```
python3 hello.py
# Output: Hello, World!
```

There are two ways to run Python: saving code in a **.py file** and executing it (what you just did), or typing commands one at a time in the **interactive shell** (REPL) by simply typing `python3` in your terminal. Use the shell to experiment; use files for real programs.

> **■ CRITICAL — Indentation is not style — it is syntax**
> Python uses whitespace (indentation) to define code blocks instead of curly braces {}. Mixing tabs and spaces, or indenting inconsistently, causes an IndentationError and will crash your program. Pick spaces (4 is the convention) and stay consistent everywhere.

CHAPTER 02

Variables and Data Types

A variable is a name that points to a value in memory. You don't declare a type — you just assign.

```
age = 25 # int
price = 19.99 # float
name = "Alex" # str
is_active = True # bool
nothing = None # NoneType (absence of a value)
```

Checking a Type

```
print(type(age)) # <class 'int'>
print(type(price)) # <class 'float'>
```

Core Built-in Types

- **int** — whole numbers: 1, -4, 1000000
- **float** — decimal numbers: 3.14, -0.5
- **str** — text, wrapped in quotes: "hello" or 'hello'
- **bool** — True or False
- **list**, **tuple**, **dict**, **set** — collections, covered in chapters 5–6

Naming Rules

Variable names must start with a letter or underscore, contain only letters/numbers/underscores, and cannot be a reserved keyword (like `if` or `class`). Python is case-sensitive: `age` and `Age` are different variables. Convention: use `snake_case` for variables and functions.

> **■ CRITICAL — int vs float division**
>
> The / operator always returns a float, even for whole-number results: 10 / 2 gives 5.0, not 5. If you need integer division, use // instead: 10 // 3 gives 3 (rounds down, not to nearest).

CHAPTER 03

Operators

Arithmetic

```
7 + 3 # 10 addition
7 - 3 # 4 subtraction
7 * 3 # 21 multiplication
7 / 3 # 2.333... true division
7 // 3 # 2 floor division
7 % 3 # 1 modulo (remainder)
7 ** 3 # 343 exponent
```

Comparison (return True/False)

```
5 == 5 # True equal to
5 != 3 # True not equal
5 > 3 # True
5 <= 5 # True
```

Logical

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

Assignment Shortcuts

```
x = 5
x += 1 # same as x = x + 1 -> 6
x *= 2 # x = x * 2 -> 12
```

> **■ CRITICAL — == compares value, 'is' compares identity**
> == checks whether two values are equal. 'is' checks whether two names point to the exact same object in memory. [1,2,3] == [1,2,3] is True, but [1,2,3] is [1,2,3] is False — they're equal lists but two different objects. Always use == for value comparison; reserve 'is' for checking against None.

CHAPTER 04

Strings

Strings are immutable sequences of characters. Once created, a string's contents cannot be changed in place.

```
s = "Python"
print(s[0]) # 'P' indexing (0-based)
print(s[-1]) # 'n' negative index = from the end
print(s[0:3]) # 'Pyt' slicing [start:stop)
print(len(s)) # 6 length
```

Common Methods

```
" hi ".strip() # "hi" remove outer whitespace
"Hi".lower() # "hi"
"hi".upper() # "HI"
"a,b,c".split(",") # ['a', 'b', 'c']
",".join(["a","b","c"]) # "a,b,c"
"hi" in "python hi!" # True (membership test)
```

f-strings (formatted strings) — the modern standard

```
name = "Alex"
age = 25
print(f"{name} is {age} years old")
# Alex is 25 years old
```

f-strings let you embed any expression inside {} and even format numbers directly:

```
pi = 3.14159
print(f"{pi:.2f}") # 3.14 (2 decimal places)
```

> **■ CRITICAL — Strings are immutable**
> s = "cat"; s[0] = "b" raises a TypeError. You cannot modify a string in place. Instead, build a new string: s = "b" + s[1:]. This also means repeated string concatenation in a loop (s += x) is slow for large amounts of text — use "".join(list_of_pieces) instead.

CHAPTER 05

Lists and Tuples

Lists — ordered, mutable collections

```
fruits = ["apple", "banana", "cherry"]
fruits.append("date") # add to end
fruits[0] = "apricot" # mutate in place
fruits.remove("banana") # remove by value
fruits.pop() # remove & return last item
print(fruits) # ['apricot', 'cherry']
print(len(fruits)) # 2
```

Slicing

```
nums = [0,1,2,3,4,5]
print(nums[1:4]) # [1, 2, 3]
print(nums[:3]) # [0, 1, 2]
print(nums[::-1]) # [5, 4, 3, 2, 1, 0] reversed
```

Tuples — ordered, immutable collections

Tuples look like lists but cannot be changed after creation. Use them for fixed collections (coordinates, RGB values) where accidental mutation would be a bug.

```
point = (3, 4)
x, y = point # unpacking
print(x, y) # 3 4
```

> **■ CRITICAL — A one-item tuple needs a trailing comma**
> (1) is just the integer 1 in parentheses, not a tuple. You must write (1,). This trips up nearly every beginner at least once.

CHAPTER 06

Dictionaries and Sets

Dictionaries — key/value pairs

```
person = {"name": "Alex", "age": 25}
print(person["name"]) # 'Alex'
person["email"] = "a@x.com" # add a new key
person.get("phone", "N/A") # 'N/A' safe lookup, no crash
for key, value in person.items():
print(key, "->", value)
```

Sets — unordered, unique values

```
a = {1, 2, 3}
b = {2, 3, 4}
print(a | b) # union {1, 2, 3, 4}
print(a & b) # intersection {2, 3}
print(a - b) # difference {1}
```

Sets automatically remove duplicates — a fast way to de-duplicate a list is `list(set(my_list))` (note: this loses original order).

> **■ CRITICAL — Dictionary keys must be immutable**
> You can use a string, number, or tuple as a dict key, but not a list: {[1,2]: "x"} raises TypeError: unhashable type: 'list'. If you need a list-like key, convert it to a tuple first.

CHAPTER 07

Conditionals

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

Truthy and Falsy Values

Every value in Python can be tested as True or False in an if-statement. The following are all falsy: 0, 0.0, "" (empty string), [] (empty list), {} (empty dict), set(), and None. Everything else is truthy.

```
items = []
if items:
print("has items")
else:
print("empty") # this runs
```

Ternary (inline if)

```
status = "adult" if age >= 18 else "minor"
```

> **■ CRITICAL — Don't compare to True/False/None with ==**
> Write 'if is_valid:' not 'if is_valid == True:'. And always use 'if x is None:' rather than 'if x == None:' — 'is' is the correct, idiomatic check for None and avoids edge cases with custom objects that override ==.

CHAPTER 08

Loops

for loops — iterate over a sequence

```
for fruit in ["apple", "banana", "cherry"]:
print(fruit)

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

for i, fruit in enumerate(["apple", "banana"]):
print(i, fruit) # 0 apple / 1 banana
```

while loops — repeat until a condition is false

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

break, continue, else

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

> **■ CRITICAL — Beware infinite loops**
>
> A while loop runs forever if its condition never becomes False. Always make sure something inside the loop moves it toward the exit condition (e.g. count += 1). If your program hangs, this is the first thing to check.

CHAPTER 09

Functions

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

print(greet("Alex")) # Hello, Alex!
print(greet("Sam", "Hi")) # Hi, Sam!
print(greet(name="Kim")) # keyword argument
```

Why functions matter

Functions let you name a piece of logic once and reuse it, instead of copy-pasting code. They take **parameters** (inputs) and can **return** a value (output). A function with no return statement returns None by default.

*args and **kwargs

```
def total(*numbers):
return sum(numbers)
total(1, 2, 3) # 6, any number of positional args

def describe(**info):
for k, v in info.items():
print(k, v)
describe(name="Alex", age=25) # any number of keyword args
```

> **■ CRITICAL — Never use a mutable object as a default argument**
> def add_item(item, bag=[]): looks reasonable but is a classic bug. The default list is created ONCE when the function is defined, not each call — every call without a bag argument shares and mutates the same list, so items pile up across unrelated calls. Fix: use bag=None, then inside the function write 'if bag is None: bag = []'.

CHAPTER 10

Error Handling

Errors ('exceptions') stop your program unless you catch them with try/except.

```
try:
result = 10 / 0
except ZeroDivisionError:
print("Can't divide by zero")
except Exception as e:
print(f"Something else went wrong: {e}")
else:
print("Only runs if no exception occurred")
finally:
print("Always runs, error or not")
```

Raising Your Own Errors

```
def set_age(age):
if age < 0:
raise ValueError("age cannot be negative")
return age
```

Common Built-in Exceptions

- **ValueError** — right type, wrong value (int("abc"))
- **TypeError** — wrong type used for an operation ("2" + 2)
- **KeyError** — missing dictionary key
- **IndexError** — list index out of range
- **FileNotFoundError** — file doesn't exist on disk

> **■ CRITICAL — Never use a bare 'except:'**
> except: (with no exception type) silently catches everything — including KeyboardInterrupt and typos in your own code — making bugs nearly impossible to find. Always name the exception you expect, e.g. except ValueError:, or at minimum except Exception as e: and log e.

CHAPTER 11

Files: Reading and Writing

```
# Writing
with open("notes.txt", "w") as f:
f.write("Hello file!\n")

# Reading
with open("notes.txt", "r") as f:
content = f.read()
print(content)

# Reading line by line
with open("notes.txt", "r") as f:
for line in f:
print(line.strip())
```

File Modes

- **"r"** read (default, file must exist)
- **"w"** write (creates file, ERASES existing content)
- **"a"** append (adds to the end, keeps existing content)

> **■ CRITICAL — Always use 'with open(...)'**
> with open(...) as f: automatically closes the file when the block ends, even if an error occurs inside it. If you call open() directly without 'with' and forget f.close(), you risk file corruption, resource leaks, or 'too many open files' errors in long-running programs.

CHAPTER 12

Modules, Packages & pip

A module is just a .py file you can import. Python's standard library ships with dozens of ready-to-use modules; third-party packages are installed with pip.

```
import math
print(math.sqrt(16)) # 4.0

from datetime import datetime
print(datetime.now())

import random as rnd
print(rnd.choice(["a", "b", "c"]))
```

Installing Third-Party Packages

```
pip install requests
# then in code:
import requests
```

Your Own Modules

```
# math_utils.py
def square(x):
return x * x

# main.py
from math_utils import square
print(square(5)) # 25
```

> **■ CRITICAL — Use virtual environments**
> Installing packages globally causes version conflicts between projects. Create an isolated environment per project: python3 -m venv venv, then activate it (source venv/bin/activate on macOS/Linux) before running pip install. This is standard professional practice, not optional.

CHAPTER 13

Object-Oriented Programming

A class is a blueprint for creating objects that bundle data (attributes) and behavior (methods).

```
class Dog:
def __init__(self, name, breed):
self.name = name
self.breed = breed

def bark(self):
return f"{self.name} says Woof!"

rex = Dog("Rex", "Labrador")
print(rex.bark()) # Rex says Woof!
print(rex.breed) # Labrador
```

self

self refers to the specific object the method is being called on. It is passed automatically by Python; you never pass it yourself when calling rex.bark().

Inheritance

```
class Puppy(Dog):
def bark(self):
return f"{self.name} says Yip!" # overrides parent method

p = Puppy("Milo", "Beagle")
print(p.bark()) # Milo says Yip!
```

> **■ CRITICAL — __init__ is not a constructor, it's an initializer**
> The object already exists by the time __init__ runs (Python calls __new__ first, which you'll rarely touch). __init__ just sets up the object's initial state. Forgetting 'self' as the first parameter of any method is the single most common beginner error in OOP code — it causes a TypeError about the wrong number of arguments.

CHAPTER 14

Comprehensions & Lambda

A comprehension builds a new list, dict, or set in one readable line instead of a multi-line loop.

```
squares = [x**2 for x in range(6)]
# [0, 1, 4, 9, 16, 25]

evens = [x for x in range(10) if x % 2 == 0]
# [0, 2, 4, 6, 8]

square_map = {x: x**2 for x in range(4)}
# {0: 0, 1: 1, 2: 4, 3: 9}
```

Lambda — small, anonymous functions

```
double = lambda x: x * 2
print(double(5)) # 10

names = ["charlie", "alice", "bob"]
names.sort(key=lambda n: n)
print(names) # ['alice', 'bob', 'charlie']
```

> **■ CRITICAL — Readability beats cleverness**
> A comprehension nested more than one level deep, or a lambda doing complex logic, is harder to debug than a plain for-loop or named function. If you have to squint to read it, rewrite it as a regular loop.

CHAPTER 15

Scope, Copies & Critical Gotchas

Local vs Global Scope

```
x = 10
def change():
x = 20 # creates a NEW local x, does not touch global x
print(x) # 20
change()
print(x) # still 10
```

To modify a global variable inside a function, you must explicitly declare it with the `global` keyword — but doing so is usually a sign your code needs restructuring.

Shallow Copy vs Deep Copy

```
import copy
original = [[1, 2], [3, 4]]
shallow = original.copy() # or list(original)
shallow[0][0] = 99
print(original) # [[99, 2], [3, 4]] <- inner list was shared!

deep = copy.deepcopy(original)
deep[0][0] = 1
print(original) # unaffected
```

■ CRITICAL — Small integers and short strings are cached — don't rely on 'is'

Python pre-caches small integers (-5 to 256) and some strings, so 'a = 5; b = 5; a is b' can print True even though 'is' checks identity, not value. This is an implementation detail, not a language guarantee. Larger numbers behave differently: a = 1000; b = 1000; a is b is often False. Never use 'is' for value comparison — always use ==.

■ CRITICAL — Floating-point numbers are not exact

0.1 + 0.2 == 0.3 evaluates to False, because floats are stored in binary and 0.1 cannot be represented exactly. Never compare floats with ==. Instead check the difference: abs(a - b) < 1e-9, or use the decimal module for money and other exact-precision needs.

CHAPTER 16

Working With the Standard Library

Python ships with a huge standard library. A few modules you'll use constantly:

```
import os
os.listdir(".") # list files in current directory

import json
data = json.loads('{"a": 1}') # parse JSON text into a dict
json.dumps(data) # convert dict back into JSON text

import datetime
today = datetime.date.today()

import random
random.randint(1, 10) # random int between 1 and 10 inclusive
```

Beyond the Standard Library

- **requests** — HTTP requests to APIs and websites
- **pandas** — data analysis and spreadsheets
- **numpy** — fast numerical arrays and math
- **flask / django** — building web applications
- **pytest** — writing automated tests

CHAPTER 17

Mini Project: Command-Line To-Do App

This project combines everything so far: functions, lists, dicts, loops, conditionals, file I/O, and error handling.

```
import json
import os

FILE = "tasks.json"

def load_tasks():
if not os.path.exists(FILE):
return []
with open(FILE, "r") as f:
return json.load(f)

def save_tasks(tasks):
with open(FILE, "w") as f:
json.dump(tasks, f, indent=2)

def add_task(tasks, text):
tasks.append({"text": text, "done": False})
save_tasks(tasks)

def complete_task(tasks, index):
try:
tasks[index]["done"] = True
save_tasks(tasks)
except IndexError:
print("No such task.")

def show_tasks(tasks):
for i, t in enumerate(tasks):
mark = "x" if t["done"] else " "
print(f"[{mark}] {i}: {t['text']}")

if __name__ == "__main__":
tasks = load_tasks()
add_task(tasks, "Learn Python")
add_task(tasks, "Build a project")
complete_task(tasks, 0)
show_tasks(tasks)
```

todo.py

Notice the pattern: small, single-purpose functions (load, save, add, complete, show), each doing one job, combined by a main block. This is how real Python programs are structured — not as one giant script, but as small pieces that compose.

CHAPTER 18

Cheat Sheet

Task	Syntax
Define a variable	`x = 5`
Function	`def f(x): return x * 2`
If/else	`if x > 0: ... else: ...`
For loop	`for i in range(n): ...`
While loop	`while cond: ...`
List	`[1, 2, 3]`
Dict	`{"key": "value"}`
List comprehension	`[x*2 for x in items]`
Try/except	`try: ... except ValueError: ...`
Open file	`with open("f.txt") as f: ...`
Class	`class C:\n  def __init__(self): ...`
Import	`import module  /  from m import x`
f-string	`f"{value:.2f}"`
Lambda	`lambda x: x + 1`

CHAPTER 19

Common Beginner Mistakes

- **Forgetting the colon** at the end of if/for/while/def/class lines — causes a SyntaxError.
- **Off-by-one errors** with range() — range(5) is 0,1,2,3,4, not 1–5.
- **Confusing = and ==** — = assigns, == compares. if x = 5: is a SyntaxError (Python protects you here).
- **Modifying a list while looping over it** — skips or duplicates elements. Loop over a copy: for x in list[:]:
- **Comparing floats with ==** — see Chapter 15.
- **Not understanding mutable default arguments** — see Chapter 9.
- **Catching every exception with a bare except:** — see Chapter 10.
- **Mixing tabs and spaces** — see Chapter 1.
- **Using is instead of ==** for value comparisons — see Chapter 3 and 15.
- **Not closing files** — always use 'with open(...)'.

CHAPTER 20

Where to Go Next

You now have the full foundation: syntax, data structures, control flow, functions, error handling, files, modules, and OOP. From here, depth comes from building things.

Suggested Path

- Build 3–5 small projects (CLI tools, simple games, scripts that automate a boring task you actually do).
- Learn **git** for version control — non-negotiable for any real project.
- Pick a direction: **web** (Flask/Django), **data** (pandas/numpy), **automation** (scripting, APIs), or **general software**.
- Read other people's code on GitHub in your direction of interest.
- Write tests for your code with **pytest** — this is what separates hobby code from professional code.
- Re-read Chapters 10, 15, and 19 of this course after your first real bug — they will mean more the second time.

End of course. The fastest way to get better from here is to build something you actually want to exist.