PYTHON

Advanced

Advanced OOP · Iterators & Context Managers · Concurrency
Async Python · Type Hints & Static Typing · Packaging & Distribution
Performance · Capstone: A Real CLI Tool

Philip Osztromok

Generated with Claude

Table of Contents

Python Advanced — 8 Chapters

 Advanced OOP

Welcome to Python Advanced. Course 2 covered classes, inheritance, and dunder methods. This chapter pushes further into territory unique to Python's object system: abstract base classes that enforce a contract on subclasses, multiple inheritance and the Method Resolution Order that makes it deterministic, and a short, honest introduction to metaclasses — one of Python's more advanced (and rarely-needed) features.

📜 Abstract Base Classes

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
def area(self):
        pass # no implementation — subclasses MUST provide one
class Square(Shape):
    def __init__(self, side):
        self.side = side

    def area(self):
        return self.side ** 2
Shape()     # TypeError: Can't instantiate abstract class Shape with abstract method area
Square(4).area()   # 16 — fine, Square actually implements area()
```

An **abstract base class** (inheriting from `ABC`) can declare methods with `@abstractmethod` that have no real implementation — just a contract. Python enforces this at instantiation time: you literally cannot create a `Shape()` instance, and any subclass that fails to override `area()` can't be instantiated either. This is stronger than Course 2, Chapter 2's plain `Shape` base class (which just returned `0` and quietly allowed instantiation) — here, forgetting to implement `area()` is caught immediately, not silently wrong.

🧬 Multiple Inheritance

```
class Flyer:
    def move(self):
        return "flying"
class Swimmer:
    def move(self):
        return "swimming"
class Duck(Flyer, Swimmer):   # inherits from BOTH
pass
Duck().move()   # "flying" — Flyer is listed first, so it wins
```

Unlike Course 2's single-parent inheritance, Python allows a class to inherit from multiple parents at once. When more than one parent defines the same method, Python needs a deterministic rule to decide which one actually runs — that rule is the MRO.

🔗 The Method Resolution Order (MRO)

```
print(Duck.__mro__)
# (<class 'Duck'>, <class 'Flyer'>, <class 'Swimmer'>, <class 'object'>)
```

The **Method Resolution Order** is the exact sequence Python searches through to find a method — itself, then each parent, left to right, following Python's C3 linearization algorithm. `Duck.move()` resolves to `Flyer.move` specifically because `Flyer` appears before `Swimmer` in `class Duck(Flyer, Swimmer):`. Calling `Duck.__mro__` (or `Duck.mro()`) shows the exact order any method lookup will follow.

⚠ Multiple Inheritance Gets Confusing Fast — Prefer Composition

The classic "diamond problem" — two parent classes that both inherit from a common grandparent, each overriding the same method differently — is exactly what the MRO exists to resolve deterministically. But deterministic doesn't mean easy to read: as inheritance chains get deeper, predicting which method actually runs from the class definition alone gets genuinely hard. In real code, favor **composition** (an object holding a reference to another object, and delegating to it) over multiple inheritance wherever it achieves the same goal — it's usually far easier to reason about.

🧪 A Metaclasses Intro

Just as a class is a blueprint for creating objects, a **metaclass** is a blueprint for creating classes — every class in Python is itself an instance of a metaclass, and by default, that metaclass is `type`:

```
print(type(42))          # <class 'int'> — 42 is an instance of int
print(type(Duck))        # <class 'type'> — Duck is an instance of type
class LoudMeta(type):              # a custom metaclass, inheriting from type
def __new__(mcs, name, bases, namespace):
    print(f"Creating class: {name}")
    return super().__new__(mcs, name, bases, namespace)

class Widget(metaclass=LoudMeta):  # Widget's CREATION triggers LoudMeta.__new__
pass
# Creating class: Widget — printed the moment the class body finishes, not when Widget() is called
```

A metaclass hooks into the process of *defining* a class, not creating instances of it — this runs once, when Python processes the `class Widget:` statement itself. Metaclasses are genuinely rare to write in application code; they show up mainly inside frameworks (Django's ORM uses one to turn model class attributes into database fields automatically). Recognizing what a metaclass is matters more than needing to write your own — most Python code never does.

Inheritance Models: Go vs Kotlin vs Python

Language	Approach
Go	No inheritance at all — interfaces (implicit satisfaction) plus struct embedding cover similar ground, with no diamond problem possible since there's no class hierarchy to create one.
Kotlin	Single inheritance only (one `open class` parent), but a class can implement multiple `interface`s — Kotlin sidesteps the diamond problem by only allowing ambiguity between interface default methods, which it forces you to resolve explicitly with `super<InterfaceName>`.
Python	True multiple inheritance from multiple concrete classes, resolved deterministically via the MRO — more powerful, and correspondingly easier to misuse.

Kotlin's "single class, multiple interfaces" model is a deliberate middle ground — it gets most of multiple inheritance's benefit while designing away the diamond problem's worst confusion, something Python's fuller model doesn't do for you automatically.

ABC / @abstractmethod

Enforces a contract — a class can't be instantiated until it implements every abstract method.

Multiple inheritance

`class C(A, B):` — inherits from more than one parent at once.

MRO

The deterministic order Python searches parents in — view it with `Class.__mro__`.

Metaclass

A "class of a class" — `type` by default; hooks into class creation itself.

💻 Coding Challenges

Challenge 1: Enforced Payment Interface

Write an abstract base class `PaymentMethod(ABC)` with an abstract method `process(amount)`. Write a subclass `CreditCard` that implements it (just returning a confirmation string is fine). Prove that instantiating `PaymentMethod()` directly raises a `TypeError`.

Goal: Practice defining and enforcing an abstract base class contract.

→ Solution

Challenge 2: Trace the MRO

Given three classes `A`, `B(A)`, and `C(A)`, and a fourth class `D(B, C)`, print `D.__mro__` and, in a comment, explain in one sentence why `A` appears only once despite being an ancestor of both `B` and `C`.

Goal: Get hands-on with reading an actual MRO output for a real diamond-shaped hierarchy.

→ Solution

Challenge 3: Composition Over Multiple Inheritance

Rewrite this chapter's `Duck(Flyer, Swimmer)` example using composition instead — a `Duck` class that holds a `Flyer` instance and a `Swimmer` instance as attributes, with its own `fly()` and `swim()` methods that delegate to them.

Goal: Practice the composition alternative this chapter's warn-box recommends, and see how it avoids MRO ambiguity entirely.

→ Solution

🎯 What's Next

Next chapter: **Iterators & Context Managers Deep Dive** — building a custom iterator, `contextlib`, and `__enter__`/`__exit__`.

🔁 Iterators & Context Managers Deep Dive

Course 2, Chapter 5 showed `yield` as an easy way to produce lazy sequences. This chapter goes underneath that: the iterator protocol every `for` loop actually relies on, how to implement it yourself in a class, and the same "deep dive" treatment for context managers — the real `__enter__`/`__exit__` protocol behind `with`, plus `contextlib`'s easier decorator-based shortcut.

🔁 The Iterator Protocol

Every `for` loop you've written since Course 1 is sugar for a lower-level protocol: calling `iter()` once, then `next()` repeatedly until `StopIteration` is raised:

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

# what "for n in numbers:" actually does, unrolled:
iterator = iter(numbers)
while True:
    try:
        n = next(iterator)
    except StopIteration:
        break
print(n)
```

An object is **iterable** if it defines `__iter__` (which `iter()` calls, returning an iterator). An object is an **iterator** if it defines `__next__` (which `next()` calls, returning the next value or raising `StopIteration` when exhausted). A list is iterable but is *not itself* an iterator — `iter(numbers)` returns a separate list_iterator object that actually tracks position.

🪜 Building a Custom Iterator Class

```
class Countdown:
    def __init__(self, start):
        self.current = start

    def __iter__(self):
        return self        # Countdown IS its own iterator
def __next__(self):
        if self.current <= 0:
            raise StopIteration
        self.current -= 1
return self.current + 1
for n in Countdown(3):
    print(n)
# 3 2 1
```

This is exactly what Course 2, Chapter 5's `count_up_to` generator function did automatically — a generator function is really just a much more convenient way to get an object with `__iter__` and `__next__` already implemented correctly, without writing the class yourself.

⚠ A Custom Iterator Has the Same Single-Use Limitation as a Generator

Because `Countdown.__iter__` returns `self`, and `self.current` only ever counts down, a single `Countdown` instance is exhausted after one full loop through it — iterating the same instance again yields nothing, the exact same one-pass limitation Course 2, Chapter 5 covered for generators. Create a fresh instance (`Countdown(3)` again) for a second pass, the same fix that chapter recommended for generators.

🔒 Context Managers: __enter__ and __exit__

```
class Timer:
    def __enter__(self):
        import time
        self.start = time.time()
        return self             # becomes the "as" variable
def __exit__(self, exc_type, exc_value, traceback):
        import time
        print(f"Elapsed: {time.time() - self.start:.2f}s")
        return False # False = don't suppress any exception — see the warning below
with Timer() as t:
    sum(range(10_000_000))
# Elapsed: 0.34s
```

This is the class-based version of Course 2, Chapter 4's `with open(...) as file:` — `__enter__` runs at the start of the `with` block (its return value becomes the `as` variable), and `__exit__` always runs at the end, even if an exception occurred inside the block.

⚠ __exit__ Returning True Silently Swallows the Exception

`__exit__` receives details about any exception that occurred inside the `with` block (`exc_type`, `exc_value`, `traceback` — all `None` if nothing went wrong). If `__exit__` returns a truthy value, Python treats the exception as **handled** and suppresses it entirely — the code after the `with` block keeps running as if nothing happened. This is almost never what you want; return `False` (or nothing, which is falsy) unless you deliberately intend to swallow specific exceptions.

🎁 contextlib.contextmanager

```
from contextlib import contextmanager
import time

@contextmanager
def timer():
    start = time.time()
    try:
        yield # the "with" block's code runs here
finally:
        print(f"Elapsed: {time.time() - start:.2f}s")

with timer():
    sum(range(10_000_000))
# Elapsed: 0.34s — same result, far less code than the Timer class
```

`@contextmanager` turns a generator function into a context manager: code before `yield` becomes `__enter__`'s logic, code after `yield` (inside `finally`, guaranteeing it runs) becomes `__exit__`'s logic. For simple cases, this is noticeably less boilerplate than writing a full class with both dunder methods — the same trade-off decorators (Course 2, Chapter 6) generally offer over writing the underlying mechanism by hand.

Custom Iteration: Go vs Kotlin vs Python

Language	Approach
Go	No built-in iterator protocol until relatively recently (range-over-func) — historically, channels combined with goroutines were the idiomatic way to produce a lazy sequence.
Kotlin	An `Iterator<T>` interface with `hasNext()` and `next()` — structurally very close to Python's `__next__`/`StopIteration` pair, just with an explicit boolean check instead of an exception.
Python	`__iter__`/`__next__`, with `StopIteration` as the "no more values" signal instead of a boolean check.

Kotlin's explicit `hasNext()` check versus Python's exception-based `StopIteration` is a real philosophical difference worth noticing — Python leans on "ask forgiveness, not permission" here, the same style choice that shows up in its exception-handling culture generally (Course 2, Chapter 3).

`__iter__` / `__next__`

The protocol every `for` loop relies on; `next()` raises `StopIteration` when done.

Iterable vs iterator

Iterable has `__iter__`; an iterator additionally has `__next__`.

`__enter__` / `__exit__`

The class-based protocol behind `with`; `__exit__` must return falsy to not suppress exceptions.

@contextmanager

Turns a `yield`-based generator function into a context manager, less boilerplate than a class.

💻 Coding Challenges

Challenge 1: An Evens-Only Iterator

Write a class `EvensUpTo` implementing `__iter__` and `__next__`, that iterates only the even numbers from 0 up to (and including) a given limit. Use it in a `for` loop with a limit of 10.

Goal: Practice implementing the iterator protocol directly in a class, reusing this chapter's `Countdown` structure.

→ Solution

Challenge 2: A Class-Based Logging Context Manager

Write a class `LogSection` that takes a `name` in `__init__`, prints `f"Entering {name}"` in `__enter__`, and prints `f"Exiting {name}"` in `__exit__`. Use it in a `with` block around a couple of print statements.

Goal: Practice writing `__enter__`/`__exit__` directly, reusing this chapter's `Timer` structure.

→ Solution

Challenge 3: The Same Logging Context Manager with contextlib

Rewrite Challenge 2's `LogSection` as a `@contextmanager`-decorated generator function called `log_section(name)` instead, producing the same "Entering .../Exiting ..." output.

Goal: Directly compare the class-based and `@contextmanager`-based approaches to the exact same problem.

→ Solution

🎯 What's Next

Next chapter: **Concurrency** — `threading`, `multiprocessing`, and the GIL explained.

Concurrency

Python has a genuinely unusual constraint most languages don't: the Global Interpreter Lock. This chapter explains what the GIL actually is and why it exists, then covers the two standard-library tools for concurrent work — `threading` and `multiprocessing` — and, critically, when each one actually helps given the GIL's constraint.

🔒 The GIL Explained

The **Global Interpreter Lock** (GIL) is a lock built into the standard Python interpreter (CPython) that allows only *one thread* to execute Python bytecode at a time — even on a machine with 8 CPU cores, only one core is ever running Python code at any given instant.

The GIL exists because CPython manages memory with simple reference counting (every object tracks how many references point to it; when that count hits zero, it's freed). Reference counting isn't thread-safe on its own — two threads incrementing the same count simultaneously could corrupt it. The GIL sidesteps that entire class of bug by simply never letting two threads run Python code at once, which is far simpler than the fine-grained locking a fully thread-safe interpreter would need.

threading

```
import threading
import time

def download(name):
    print(f"{name}: starting")
    time.sleep(2)   # simulates waiting on a slow network response
print(f"{name}: done")

threads = [threading.Thread(target=download, args=(f"file{i}",)) for i in range(3)]

for t in threads:
    t.start()
for t in threads:
    t.join()   # wait for every thread to finish before continuing
# total time: ~2 seconds, NOT 6 — all three downloads overlap
```

This is where threading genuinely helps despite the GIL: `time.sleep()` (and real I/O like network requests or file reads) **releases the GIL** while waiting, letting another thread run Python code during that wait. Three 2-second downloads run in about 2 seconds total, not 6 — the threads overlap during their idle waiting time, even though only one is ever actively running Python bytecode at once.

⚠ Shared State Across Threads Still Needs a Lock

Even with the GIL, operations that take multiple steps internally (like `counter += 1`, which reads, adds, then writes back) aren't guaranteed atomic — two threads can interleave mid-operation and lose an update, a classic race condition. `threading.Lock()` (used as a context manager, tying directly back to Chapter 2's `with` coverage) protects a shared resource by ensuring only one thread touches it at a time: `with lock: counter += 1`.

🖥️ multiprocessing

```
import multiprocessing

def compute_square(n):
    return n ** 2
if __name__ == "__main__":
    with multiprocessing.Pool(4) as pool:
        results = pool.map(compute_square, range(10))
    print(results)   # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81] — computed across 4 real processes
```

`multiprocessing` sidesteps the GIL entirely by running each task in a genuinely separate operating system process — each with its own Python interpreter, its own memory space, and crucially, its own GIL. This is how Python achieves real parallelism for CPU-heavy work (like the squaring above, or any computation-bound loop), something `threading` alone cannot do because of the single shared GIL.

🎯 When to Use Which

I/O-Bound vs CPU-Bound Work

Work type	Bottleneck	Right tool
I/O-bound	Waiting on network, disk, or a database — the CPU is mostly idle	threading
CPU-bound	Heavy computation keeping the CPU genuinely busy	multiprocessing

`threading` is "cheap" (threads share memory, start fast) but limited to one running at a time by the GIL — perfect for waiting, useless for parallel computation. `multiprocessing` is "expensive" (each process needs its own memory, starts slower, and data must be explicitly passed between processes) but gets real parallel CPU work done. Picking the wrong one for the job either wastes overhead (multiprocessing for I/O waits) or does nothing useful (threading for CPU-bound math).

Concurrency Models: Go vs Kotlin vs Python

Language	Approach
Go	Goroutines are extremely lightweight and scheduled M:N onto real OS threads by the Go runtime — no GIL equivalent, so goroutines genuinely run in parallel across cores for both I/O and CPU work.
Kotlin	Coroutines run on real JVM threads with cooperative scheduling — no GIL either, so CPU-bound coroutine work can genuinely use multiple cores, unlike Python threads.
Python	`threading` for I/O-bound concurrency only (GIL-limited); `multiprocessing` required for real CPU-bound parallelism.

The GIL is the single biggest structural difference between Python's concurrency story and Go's or Kotlin's — both of those languages simply don't have this constraint, so a single lightweight concurrency primitive covers both I/O-bound and CPU-bound cases, where Python genuinely needs two different tools for the two different problems.

The GIL

Only one thread runs Python bytecode at a time; exists to keep reference counting simple.

threading

Good for I/O-bound work — the GIL releases during waits, letting threads overlap.

multiprocessing

Separate processes, separate GILs each — real parallelism for CPU-bound work.

Locks

`threading.Lock()` protects shared state from race conditions between threads.

💻 Coding Challenges

Challenge 1: Timed Sequential vs Threaded Downloads

Using this chapter's `download(name)` function (with `time.sleep(2)`), time how long it takes to call it 3 times sequentially in a plain loop, versus running the same 3 calls via `threading.Thread`. Print both elapsed times.

Goal: See the I/O-bound threading speedup firsthand, not just read about it.

→ Solution

Challenge 2: A Race Condition, Then Fixed

Write a shared `counter = 0` incremented 100,000 times across 2 threads (50,000 increments each, via `counter += 1` with no lock) — print the final count (it likely won't be 100,000). Then fix it using `threading.Lock()` and show the count is now correct every time.

Goal: Observe a real race condition, then apply this chapter's `Lock` fix directly.

→ Solution

Challenge 3: Parallel Sum of Squares

Using `multiprocessing.Pool`, compute the square of every number from 0 to 19 in parallel across 4 processes, then sum the results and print the total.

Goal: Practice `Pool.map()` for genuinely CPU-parallel work.

→ Solution

🎯 What's Next

Next chapter: **Async Python** — `asyncio`, `async`/`await`, and the event loop.

⏱️ Async Python

Chapter 3 covered `threading` and `multiprocessing` — two ways to run work concurrently using multiple OS threads or processes. This chapter covers a third model: `asyncio`, which runs on a *single thread* but juggles many I/O-bound tasks through cooperative multitasking on an event loop, using `async/await`.

🔤 async/await Basics

```
import asyncio

async def say_hello():
    print("Hello...")
    await asyncio.sleep(1)  # a non-blocking "wait" — hands control back to the event loop
print("...world!")

coro = say_hello()  # does NOT run the function yet — just creates a coroutine object
print(type(coro))  # <class 'coroutine'>
```

`async def` defines a **coroutine function** — calling it doesn't run the body immediately, exactly like calling a generator function (Course 2, Chapter 5) doesn't run its body immediately either. It only produces a coroutine object; something has to actually *drive* that coroutine for its code to run.

⚠ Forgetting await Silently Does Nothing

```
async def main():
    say_hello()  # WRONG — creates a coroutine object and immediately discards it. Nothing runs!
```

Calling a coroutine function without `await`-ing it (or otherwise scheduling it) doesn't run its code at all — Python creates the coroutine object and, since nothing holds onto it or drives it, it's simply discarded, usually with a `RuntimeWarning: coroutine 'say_hello' was never awaited`. This is one of the most common async mistakes: always `await` a coroutine call (or pass it to `asyncio.gather()`, below) — never call it bare.

▶️ Running Async Code: asyncio.run()

```
asyncio.run(say_hello())  # the entry point — starts the event loop, runs the coroutine, then stops it
# Hello...
# ...world!  (printed ~1 second later)
```

`asyncio.run()` is the top-level entry point for async code — it creates an event loop, runs the given coroutine to completion, then shuts the loop down. It should be called once, at the very top of a program (analogous to `if __name__ == "__main__":` from Chapter 3's multiprocessing example) — not from inside other async code.

👷 Running Tasks Concurrently: asyncio.gather()

```
async def download(name):
    print(f"{name}: starting")
    await asyncio.sleep(2)
    print(f"{name}: done")

async def main():
    await asyncio.gather(
        download("file1"),
        download("file2"),
        download("file3"),
    )

asyncio.run(main())
# total time: ~2 seconds, not 6 — all three "downloads" overlap on ONE thread
```

This produces the same ~2-second result as Chapter 3's threaded download example — but using a single thread the whole time. `asyncio.gather()` schedules all three coroutines onto the event loop; whenever one hits `await asyncio.sleep(2)`, it voluntarily hands control back to the loop, which runs another coroutine in the meantime.

🔁 The Event Loop

The **event loop** is the single-threaded scheduler underneath every `asyncio` program. Unlike `threading`'s OS-level preemptive scheduling (where the operating system can interrupt a thread at any instant), `asyncio` uses **cooperative** scheduling: a coroutine only ever gives up control at an explicit `await` point. Between `await`s, a coroutine runs uninterrupted — no other coroutine can sneak in, which sidesteps threading's race-condition problem (Chapter 3) entirely, without needing a single `Lock`.

Async Models: Rust vs Node.js vs Python

Language	Approach
Rust	Futures are *lazy* — creating one does nothing until it's driven by a runtime like `tokio` (`rust3-3`), and Rust ships no built-in runtime at all, a deliberate "bring your own executor" design.
Node.js	The event loop is built directly into the runtime (`node1-1`) — there's no separate "start the loop" call needed; async code just runs, since the whole runtime is event-loop-based from the start.
Python	Coroutines, like Rust's futures, are lazy — `async def` alone does nothing until `await`ed or run — but unlike Rust, `asyncio` ships as a built-in standard-library runtime, so no third-party executor is required.

Python's laziness (a coroutine call does nothing on its own) is a genuine parallel to Rust's own lazy futures — both are a real contrast to Node.js, where the event loop is simply always running underneath everything, with no equivalent "did I forget to await this?" trap.

async def

Defines a coroutine function — calling it creates a coroutine object, doesn't run it.

await

Runs a coroutine, pausing here and handing control back to the event loop until it resolves.

asyncio.run()

The top-level entry point — starts the loop, runs one coroutine, stops the loop.

asyncio.gather()

Runs multiple coroutines concurrently on one thread via cooperative scheduling.

💻 Coding Challenges

Challenge 1: A Basic Coroutine

Write an async function `brew_coffee()` that prints `"Brewing..."`, awaits `asyncio.sleep(1)`, then prints `"Coffee ready!"`. Run it with `asyncio.run()`.

Goal: Practice the basic `async def`/`await`/`asyncio.run()` mechanics.

→ Solution

Challenge 2: Timed Sequential vs Concurrent Coroutines

Using this chapter's `download(name)` coroutine, time how long it takes to `await` it 3 times sequentially inside one `async def main()`, versus running the same 3 calls via `asyncio.gather()`. Print both elapsed times.

Goal: See the concurrency benefit of `gather()` firsthand, directly paralleling Chapter 3's own threading-vs-sequential timing challenge.

→ Solution

Challenge 3: Fix the Forgotten await

Given a coroutine function `greet(name)` that prints a greeting, write an `async def main()` that calls it WITHOUT `await` (reproducing this chapter's warn-box mistake), observe that nothing prints, then fix it.

Goal: See the "forgot to await" trap firsthand and confirm the fix.

→ Solution

🎯 What's Next

Next chapter: **Type Hints & Static Typing** — the `typing` module, `mypy`, `Protocol`, and generics.

🏷️ Type Hints & Static Typing

Course 1, Chapter 2 introduced Python as dynamically typed — variables can hold any type, checked only at runtime. This chapter covers the layer Python added on top of that: optional **type hints**, the `typing` module's building blocks, `mypy` as an external static checker, `Protocol` for structural typing, and generics.

✍️ Basic Type Hints

```
def add(a: int, b: int) -> int:
    return a + b

name: str = "Ada"
age: int = 28
```

A type hint is written as `name: Type` for a variable or parameter, and `-> Type` after a function's parameter list for its return type. This looks similar to Kotlin's own type annotations, but with one crucial difference — see the warning below.

⚠ Type Hints Are NOT Enforced at Runtime

```
def add(a: int, b: int) -> int:
    return a + b

print(add("hello", "world"))  # "helloworld" — runs FINE, no error at all!
```

Unlike Go or Kotlin, where the compiler rejects a type mismatch before the program can even run, Python's type hints are pure documentation as far as the language itself is concerned — the interpreter never checks them. `add("hello", "world")` above happily runs and string-concatenates, despite the hints clearly saying both parameters should be `int`. Catching this kind of mismatch requires a separate tool — `mypy`, covered next.

🧱 typing Module Building Blocks

```
def get_names() -> list[str]:        # a list of strings
return ["Ada", "Grace"]

def get_scores() -> dict[str, int]:   # a dict of str keys, int values
return {"Ada": 100}

def find_user(user_id: int) -> str | None:   # might return a str, or might return None
if user_id == 1:
    return "Ada"
return None
```

`list[str]`, `dict[str, int]`, and similar generic-collection hints use Course 1's own collection types (Chapter 6, Chapter 7) directly as the annotation. `X | None` (modern syntax) or `Optional[X]` (the older `typing` module form, equivalent) marks a value that might legitimately be missing — the same idea as the `Option<T>`-style types more strictly-typed languages use, but here purely advisory rather than compiler-enforced.

🔍 mypy

```
# bad_math.py
def add(a: int, b: int) -> int:
    return a + b

add("hello", "world")


# run separately, from the command line:
mypy bad_math.py

# output:
# bad_math.py:4: error: Argument 1 to "add" has incompatible type "str"; expected "int"
# bad_math.py:4: error: Argument 2 to "add" has incompatible type "str"; expected "int"
```

`mypy` (installed via `pip`, per Course 1, Chapter 9) reads a script's type hints and statically checks whether the code actually respects them — *without ever running the code*. This is what gives Python-with-hints something closer to Go's or Kotlin's compile-time safety net: run `mypy` as a separate step (often wired into CI, per the site's own CI/CD Pipelines course), catch mismatches before deployment, while the interpreter itself stays exactly as permissive as it's always been.

🦆 Protocol (Structural Typing)

```
from typing import Protocol

class Honker(Protocol):
    def honk(self) -> str: ...

class Car:                  # does NOT inherit from Honker at all
def honk(self) -> str:
        return "Beep!"
def make_it_honk(thing: Honker) -> None:
    print(thing.honk())

make_it_honk(Car())   # Beep! — works, even though Car never inherited from Honker
```

Course 3, Chapter 1's `ABC` is **nominal** typing — a class must explicitly inherit from the abstract base to count as satisfying it. `Protocol` is **structural** typing — any class with a matching `honk()` method satisfies `Honker`, whether or not it ever mentions `Honker` by name. This formalizes Python's long-standing "duck typing" culture (if it walks like a duck and honks like a car horn...) into something `mypy` can actually check statically.

🧬 Generics

```python
from typing import TypeVar

T = TypeVar("T")

def first(items: list[T]) -> T:
    return items[0]

first([1, 2, 3])      # mypy infers T = int here, so this returns int
first(["a", "b"])     # mypy infers T = str here, so this returns str
```

`TypeVar` lets a function or class be generic over whatever type it's actually given — `first()` works correctly on a `list[int]` or a `list[str]` alike, and `mypy` tracks that the return type always matches whatever `T` resolved to for that specific call, the same generic pattern Go's and Kotlin's own generics (already covered elsewhere on the site) provide, just checked externally rather than by the language itself.

Type Enforcement: Go vs Kotlin vs Python

Language	When types are checked
Go	Compile time, mandatory — a type mismatch is a compile error, the program cannot even build.
Kotlin	Compile time, mandatory — same guarantee as Go, enforced by the compiler before any code runs.
Python	Never, by the language itself — hints are optional documentation; `mypy` can check them separately, entirely opt-in, and skipping it (or running mismatched code anyway) is always possible.

This is the single biggest mindset shift in this chapter: in Go or Kotlin, "it compiled" is itself a real type-safety guarantee. In Python, "it has type hints" guarantees nothing on its own — the guarantee only exists if `mypy` (or an equivalent) is actually run, and even then, only for the code paths it checks.

Type hints

`name: Type`, `-> Type` — documentation only, not enforced by the interpreter.

mypy

A separate static checker — run it to actually catch type mismatches before runtime.

Protocol

Structural typing — matches by shape (duck typing), no inheritance required, unlike `ABC`.

TypeVar / generics

A function/class that works across types, with the relationship tracked by `mypy`.

💻 Coding Challenges

Challenge 1: Annotate a Function

Take the `fizzbuzz(n)` function from Course 2, Chapter 8's testing challenge, and add full type hints: `n` should be typed as `int`, and the return type should reflect that it always returns a `str`.

Goal: Practice adding basic type hints to an existing, already-working function.

→ Solution

Challenge 2: A Protocol for Comparable Items

Define a `Protocol` called `HasArea` with a method `area(self) -> float`. Write a function `describe(shape: HasArea) -> str` that returns `f"Area is {shape.area()}"`. Call it with an instance of the `Square` class from Course 2, Chapter 2 — it should work despite `Square` never mentioning `HasArea`.

Goal: Practice structural typing with `Protocol` and confirm it works without inheritance.

→ Solution

Challenge 3: A Generic last() Function

Using `TypeVar`, write a generic function `last(items: list[T]) -> T` that returns the last item of any list. Call it once with a `list[int]` and once with a `list[str]`, printing both results.

Goal: Practice writing a generic function with `TypeVar`, mirroring this chapter's `first()` example.

→ Solution

🎯 What's Next

Next chapter: **Packaging & Distribution** — `pyproject.toml`, setuptools/poetry, and publishing to PyPI.

📦 Packaging & Distribution

Course 1, Chapter 9 covered *consuming* packages — `pip install`, `venv`. This chapter covers the other side: turning your own code into a real, installable, distributable Python package, from project structure through `pyproject.toml` to actually publishing on PyPI.

🗂️ Project Structure

```
my_package/
├── pyproject.toml
├── README.md
├── src/
│   └── my_package/
│       ├── __init__.py
│       └── core.py
└── tests/
    └── test_core.py
```

The `src/` layout (package code inside a `src/` folder, separate from tests and config) is the modern recommended structure — it prevents accidentally importing your package from the current working directory instead of the actually-installed version, a subtle bug the older "flat" layout could hide. `__init__.py` is what makes a plain folder importable as a package at all, a detail Course 1, Chapter 9's `import` coverage didn't need to mention since it only covered *using* existing packages.

⚙️ pyproject.toml

```
# pyproject.toml
[build-system]
requires = ["setuptools>=61.0"]
build-backend = "setuptools.build_meta"

[project]
name = "my_package"
version = "0.1.0"
description = "A small example package"
dependencies = [
    "requests>=2.31.0",
]
```

`pyproject.toml` is the modern, standardized project config file — it replaced the older `setup.py` approach, and plays the same role as Node's `package.json` or Rust's `Cargo.toml`. The `[project]` table declares your package's metadata and dependencies in the exact `package==version` style from Course 1, Chapter 9's own `requirements.txt` coverage, just declared alongside the package itself rather than in a separate file.

🔨 Build Tools: setuptools vs Poetry

setuptools vs Poetry

	setuptools	Poetry
Role	The traditional, most widely used build backend — does one job, building the package	An all-in-one tool: dependency management, virtual environments, building, and publishing together
Dependency locking	No built-in lock file — needs a separate tool (like `pip-tools`) for reproducible installs	Built-in `poetry.lock`, similar to `package-lock.json` or `Cargo.lock`
Best for	Simple packages, or when you want minimal tooling and full control	Larger projects wanting one consistent tool for the whole dependency/build/publish workflow

Both read from `pyproject.toml` — they differ in scope, not in the config file format itself. `setuptools` is the safer default to learn first since it's what most existing open-source packages already use; Poetry is worth reaching for once a project's dependency management gets complex enough to want a lock file.

💻 Building and Installing Locally

```
# from inside the project's root folder, with a venv active (Course 1, Chapter 9):
pip install -e .
```

`pip install -e .` is an **editable install** — it installs your package by linking directly to your source folder, rather than copying it. Changes to your code take effect immediately, with no reinstall needed, making it the standard way to develop and test a package locally before publishing it.

🚀 Publishing to PyPI

```
# build the distributable files (a wheel + a source distribution):
python -m build

# upload to TestPyPI first — a safe practice environment, separate from the real registry:
twine upload --repository testpypi dist/*

# once verified, upload to the real PyPI:
twine upload dist/*
```

`python -m build` produces the actual distributable files in a `dist/` folder; `twine` uploads them. **TestPyPI** is a full separate instance of the registry meant exactly for practice runs — publishing there costs nothing and can't be undone by mistake on the real thing.

⚠ Package Names on PyPI Are Global and Effectively Permanent

Once a package name is taken on PyPI, it's taken — there's no reserving a name in advance, and abandoned/typo'd releases are very rarely removed. Check `pypi.org/project/<name>` before committing to a name. Also double-check what actually gets included in your built package before publishing — a stray `.env` file or credentials accidentally bundled into a public release is effectively permanent too, since older versions typically stay downloadable even after a fix.

Package Publishing: Go vs Kotlin vs Python

Language	Approach
Go	No central registry or "publish" step at all — a module is just a public Git repository; `go get` fetches directly from the repo URL and its tags.
Kotlin	Maven Central (or an equivalent) — a central registry model much like PyPI, with a similar build-then-upload workflow via Gradle/Maven.
Python	PyPI — a central registry, built and uploaded via `pyproject.toml` + `build` + `twine`.

Go's repository-is-the-package model is a genuinely different philosophy from the central-registry approach both Kotlin and Python share — no separate publishing step exists in Go at all, just tagging a commit in a public repo.

pyproject.toml

The modern standard config file — metadata, dependencies, and build backend in one place.

setuptools / Poetry

Two build-tool options — a minimal backend vs. an all-in-one workflow tool.

Editable install

`pip install -e .` — develop locally with changes taking effect immediately.

build + twine

`python -m build` creates distributables; `twine upload` publishes them to PyPI.

💻 Coding Challenges

Challenge 1: Write a Minimal pyproject.toml

Write a complete `pyproject.toml` for a package called `text_utils`, version `0.1.0`, that depends on `requests>=2.28.0`, using `setuptools` as the build backend.

Goal: Practice writing the `[build-system]` and `[project]` tables correctly from scratch.

→ Solution

Challenge 2: Design a src/ Layout

Sketch out (as plain text, a folder tree — no code needed to run) the `src/` layout for a package called `calculator`, containing two modules, `basic.py` and `advanced.py`, plus one test file for each.

Goal: Get comfortable with the standard `src/` project structure this chapter recommends.

→ Solution

Challenge 3: Trace the Publishing Steps

List, in the correct order, every command-line step needed to go from a finished local package to it being live and installable from the real PyPI — including the safe practice step this chapter recommends before the real upload.

Goal: Solidify the full build-and-publish workflow as a sequence, not just isolated commands.

→ Solution

🎯 What's Next

Next chapter: **Performance** — profiling with `cProfile`, a Cython/PyPy overview, and common optimization patterns.

🚀 Performance

Python trades raw speed for developer productivity by design — but "slow" is relative, and most Python programs are never actually CPU-bound in the way that matters. This chapter covers profiling first (finding the real bottleneck before touching anything), common optimization patterns once you've found one, and a brief, honest overview of Cython and PyPy for the cases that genuinely need more.

🔬 Profiling with cProfile

```
# from the command line, profiling an entire script:
python -m cProfile -s cumulative script.py


# or programmatically, profiling just one function:
import cProfile

def slow_function():
    total = sum(n ** 2 for n in range(1_000_000))
    return total

cProfile.run("slow_function()")


# output (abbreviated):
#         3 function calls in 0.089 seconds
#   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
#        1    0.089    0.089    0.089    0.089 script.py:3(slow_function)
```

`ncalls` is how many times a function was called; `tottime` is time spent inside that function alone (excluding calls it made to other functions); `cumtime` is total time including those nested calls. Sorting by `cumtime` (as `-s cumulative` does) quickly surfaces which function is actually responsible for the most total time — the real bottleneck, rather than a guess.

⚠ Never Guess at a Bottleneck — Always Profile First

Intuition about which part of a program is "the slow part" is wrong astonishingly often — the real cost is frequently somewhere unglamorous and easy to overlook, like a function called thousands of times inside a loop rather than one dramatic-looking computation. Optimizing code without profiling first risks spending real effort making an already-fast piece of code marginally faster while the actual bottleneck goes untouched. Profile first, every time — it's often described as "premature optimization is the root of all evil," and the fix is simply: measure, don't assume.

⚡ Common Optimization Patterns

```
# prefer built-ins (implemented in C) over manual Python loops:
total = sum(numbers)            # fast
total = 0
for n in numbers: total += n   # noticeably slower for large inputs
# use a set for membership tests, not a list (Course 1, Chapter 6):
if item in allowed_set:    # O(1) — fast regardless of size
if item in allowed_list:   # O(n) — scans the whole list every time
# cache expensive repeated calls (Course 2, Chapter 6):
@lru_cache
def expensive_lookup(key):
    ...
```

Most real Python speedups come from reaching for the right tool rather than micro-tuning syntax: `sum()`/`max()`/`min()` and other built-ins run their loop in C, not interpreted Python bytecode; a `set`'s `in` check is a hash lookup instead of a linear scan; `@lru_cache` (Course 2, Chapter 6) eliminates redundant work entirely rather than making it faster. Each of these is a genuine "use the right data structure/tool," not a clever trick.

Cython Overview

```
# a .pyx file — Python syntax, with optional C-style type declarations:
def sum_squares(int n):
    cdef int i, total = 0
for i in range(n):
        total += i * i
    return total
```

Cython is a superset of Python that compiles to C — you write mostly ordinary Python, optionally adding static type declarations (`cdef int i`) in the hottest sections, and Cython compiles those parts down to genuine C-speed code. It's a targeted tool: used for the specific hot loop or numeric kernel a profiler identified as the real bottleneck, not for rewriting an entire application.

⚙️ PyPy Overview

PyPy is an alternative Python interpreter (a drop-in replacement for the standard CPython interpreter) with a Just-In-Time (JIT) compiler — it watches which code paths actually run repeatedly and compiles those to machine code on the fly, often making long-running pure-Python programs several times faster with zero code changes. The trade-off: PyPy's compatibility with C extensions (many popular data-science and numerics libraries) has historically lagged behind CPython, so it's not always a safe drop-in for every project.

Baseline Speed: Go/Rust vs Python

Language	Why it starts faster
Go / Rust	Compiled ahead of time to native machine code, with static types checked before the program ever runs — no interpreter overhead, no runtime type-checking cost.
Python	Interpreted, dynamically typed by default (Course 1, Chapter 2; Chapter 5 of this course) — every operation carries interpreter and type-checking overhead Go/Rust simply don't have.

This gap is real and expected — it's the trade Python makes for its flexibility and development speed. The right response is never "avoid Python for anything performance-sensitive," but rather: profile, optimize the specific hot path (often with the patterns above, or Cython for the truly critical inner loop), and accept that most of a typical program was never the bottleneck to begin with.

cProfile

Finds the real bottleneck — `ncalls`, `tottime`, `cumtime` per function.

Reach for the right tool

Built-ins, sets for membership, `@lru_cache` — usually bigger wins than micro-tuning.

Cython

Compile hot Python code to C, with optional static type declarations.

PyPy

A JIT-compiled alternative interpreter — often faster, with some C-extension trade-offs.

💻 Coding Challenges

Challenge 1: Profile Two Approaches

Write two functions that each sum the squares of numbers from 0 to 999,999 — one using a manual `for` loop with `+=`, one using `sum()` with a generator expression (Course 2, Chapter 5). Profile both with `cProfile.run()` and compare their `tottime`.

Goal: See the built-in-vs-manual-loop performance difference measured directly, not just asserted.

→ Solution

Challenge 2: List vs Set Membership, Timed

Build a list and a set, each containing the numbers 0 to 99,999. Time how long 10,000 `in` checks take against the list, versus the same 10,000 checks against the set, using `time.time()`.

Goal: Directly measure the O(n) vs O(1) membership-test difference this chapter describes.

→ Solution

Challenge 3: Identify the Bottleneck

Given a function `process()` that calls `slow_helper()` 1,000 times in a loop, and `slow_helper()` itself does a small, fast computation but is simply called very often — write the `cProfile.run("process()")` call needed to find this, and explain in a comment which output column (`ncalls`, `tottime`, or `cumtime`) would make this specific pattern (many cheap calls, not one expensive one) obvious.

Goal: Practice reading profiler output to distinguish "called too often" from "individually slow," a real and common distinction.

→ Solution

🎯 What's Next

Next chapter: **Capstone: Building a Real CLI Tool** — combining OOP, type hints, packaging, and testing into one small real tool, closing out Python Advanced.

🎬 Capstone: Building a Real CLI Tool

The final chapter of Python Advanced — and of the whole structured Python track. We're building **tasks**, a small real command-line task tracker: add, list, complete, and delete tasks, persisted to a JSON file, installable as a real command. It combines OOP and type hints (Course 3, Chapters 1 & 5), JSON and file handling (Course 2, Chapters 4 & 7), packaging (Chapter 6), and testing (Course 2, Chapter 8) into one working tool — deliberately mirroring the Rust track's own capstone (`rust3-6`), a task-tracker CLI built from the equivalent Rust pieces.

📋 The Data Model

```python
# src/tasks/models.py
from dataclasses import dataclass

@dataclass
class Task:
    id: int
    title: str
    done: bool = False
```

`@dataclass` is a small, genuinely useful payoff of Course 2, Chapter 2's dunder methods: instead of hand-writing `__init__`, `__repr__`, and `__eq__` the way that chapter's `Point` class did manually, `@dataclass` generates all three automatically from the type-hinted attributes above — `id: int`, `title: str`, `done: bool = False` reuse this course's own Chapter 5 type-hint syntax directly, now doing real work rather than just documentation.

💾 Storage: A TaskStorage Class

```python
# src/tasks/storage.py
import json
from pathlib import Path
from .models import Task

class TaskStorage:
    def __init__(self, path: Path):
        self.path = path

    def load(self) -> list[Task]:
        if not self.path.exists():
            return []
        data = json.loads(self.path.read_text())
        return [Task(**item) for item in data]

    def save(self, tasks: list[Task]) -> None:
        data = [task.__dict__ for task in tasks]
        self.path.write_text(json.dumps(data, indent=2))
```

`TaskStorage` composes a `Path` (Course 2, Chapter 4) rather than inheriting from anything — the same composition-over-inheritance instinct Course 3, Chapter 1 recommended. `load()` reuses `pathlib`'s `.exists()`/`.read_text()` and `json.loads()` (Course 2, Chapter 7) together; `Task(**item)` unpacks each JSON object's keys directly as keyword arguments into the dataclass. `save()` mirrors it with `.write_text()` and `json.dumps()`.

⌨️ The CLI Interface: argparse

```python
# src/tasks/cli.py
import argparse
from pathlib import Path
from .storage import TaskStorage
from .models import Task

def main():
    parser = argparse.ArgumentParser(prog="tasks")
    subparsers = parser.add_subparsers(dest="command", required=True)

    add_parser = subparsers.add_parser("add")
    add_parser.add_argument("title")

    subparsers.add_parser("list")

    complete_parser = subparsers.add_parser("complete")
    complete_parser.add_argument("id", type=int)

    args = parser.parse_args()
    storage = TaskStorage(Path.home() / ".tasks.json")
    tasks = storage.load()

    if args.command == "add":
        next_id = max((t.id for t in tasks), default=0) + 1
        tasks.append(Task(id=next_id, title=args.title))
        storage.save(tasks)
        print(f"Added task {next_id}: {args.title}")

    elif args.command == "list":
        for t in tasks:
            status = "x" if t.done else " "
print(f"[{status}] {t.id}: {t.title}")

    elif args.command == "complete":
        for t in tasks:
            if t.id == args.id:
                t.done = True
        storage.save(tasks)
        print(f"Completed task {args.id}")
```

`argparse` — a new tool for this course, part of the standard library — parses command-line arguments into a structured `args` object. `add_subparsers` gives each subcommand (`add`, `list`,

`complete`) its own arguments; `args.command` then drives a plain `if`/`elif` chain, the same branching from Course 1, Chapter 3, dispatching to whichever behavior the user asked for.

📦 Packaging It Up

```
# pyproject.toml
[build-system]
requires = ["setuptools>=61.0"]
build-backend = "setuptools.build_meta"

[project]
name = "tasks"
version = "0.1.0"

[project.scripts]
tasks = "tasks.cli:main"
```

This is Chapter 6's `pyproject.toml`, with one new piece: `[project.scripts]` registers a **console entry point** — after `pip install -e .`, typing `tasks add "Buy milk"` at the command line runs `main()` from `tasks/cli.py` directly, no `python` prefix needed. This is what turns a script into a genuine installed command.

🧪 Testing the Tool

```python
# tests/test_storage.py
from tasks.models import Task
from tasks.storage import TaskStorage

def test_save_and_load_round_trip(tmp_path):
    storage = TaskStorage(tmp_path / "tasks.json")
    original = [Task(id=1, title="Buy milk")]

    storage.save(original)
    loaded = storage.load()

    assert loaded == original

def test_load_missing_file_returns_empty_list(tmp_path):
    storage = TaskStorage(tmp_path / "does_not_exist.json")
    assert storage.load() == []
```

`tmp_path` is a built-in `pytest` fixture (Course 2, Chapter 8) — no `@pytest.fixture` needed to define it, `pytest` provides it automatically, giving each test a fresh temporary directory so tests never touch a real `~/.tasks.json` file. `loaded == original` works cleanly on the very first try because `@dataclass` already generated `__eq__` — the exact custom equality Course 2, Chapter 2 wrote by hand for `Fraction`, here provided for free.

Task Tracker: Python vs Rust (rust3-6)

Piece	Rust (`rust3-6`)	Python (this chapter)
Data model	A `Task` struct with `#[derive(...)]` macros	A `Task` dataclass with `@dataclass`
Storage abstraction	A `Storage` trait, implemented for a JSON backend	A `TaskStorage` class, composing a `Path`
Serialization	`serde` crate	`json` module (standard library)
CLI parsing	`clap` crate	`argparse` (standard library)
Testing	`cargo test`, `#[test]`	`pytest`, `test_*` functions

The shape of the two capstones is genuinely the same problem solved with each language's own idioms — Rust reaches for third-party crates (`serde`, `clap`) for JSON and CLI parsing where Python's standard library already covers both natively, a real, concrete instance of the "batteries included" difference this whole course has touched on since Chapter 6.

@dataclass

Auto-generates `__init__`/`__repr__`/`__eq__` from type-hinted attributes.

Composition

`TaskStorage` holds a `Path`, rather than inheriting from anything.

argparse subcommands

`add_subparsers()` gives each command (`add`/`list`/`complete`) its own arguments.

tmp_path fixture

A built-in `pytest` fixture — isolates file-touching tests automatically.

💻 Coding Challenges

Challenge 1: Add a delete Command

Extend the `argparse` setup with a `delete` subcommand taking an `id`, and the matching `elif` branch that removes the task with that ID from the list (using a list comprehension, Course 1/2) before calling `storage.save()`.

Goal: Practice extending a real, structured CLI tool with a new command, following the existing pattern.

→ Solution

Challenge 2: Test the complete Behavior

Write a `pytest` test using `tmp_path` that saves two tasks, "completes" one by directly mutating its `done` attribute and saving again, reloads, and asserts exactly one task now has `done == True`.

Goal: Practice testing behavior beyond a simple round trip, reusing the `tmp_path` fixture pattern.

→ Solution

Challenge 3: A Custom Exception for a Missing Task

Define a custom exception `TaskNotFoundError(Exception)` (Course 2, Chapter 3). Write a method `TaskStorage.find(tasks, task_id)` that returns the matching `Task` or raises `TaskNotFoundError` with a descriptive message if no task has that ID.

Goal: Tie the capstone back to this course's own exception-handling chapter with a realistic use case.

→ Solution

🎓 Course Complete!

That's all 8 chapters of **Python Advanced** — advanced OOP, iterators and context managers, concurrency, async, type hints, packaging, performance, and this capstone — and with it, the full three-course Python track (Fundamentals → Intermediate → Advanced, 25 chapters). **Python Projects** (`py4`) remains outlined as a lighter, project-based fourth course whenever you're ready to continue.