PYTHON

Intermediate

Object-Oriented Programming · OOP Deep Dive · Exception Handling
File I/O · Comprehensions & Generators · Decorators
Working with Data · Testing with pytest

Philip Osztromok

Generated with Claude

Table of Contents

Python Intermediate — 8 Chapters

🏛️ Object-Oriented Programming

Course 1 worked entirely with Python's built-in types — strings, lists, dicts, and so on. This chapter starts Python Intermediate by defining your *own* types: classes, the `__init__` constructor, the `self` parameter, and the difference between instance attributes and class attributes.

🧱 Defining a Class

```
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!
```

`__init__` is the constructor — it runs automatically when `Dog(...)` is called, setting up whatever data a new object needs. By convention, class names use `PascalCase`, distinguishing them at a glance from the `snake_case` functions and variables you've used since Course 1.

👤 self Explained

`self` refers to the specific object a method is being called on — it's always the first parameter of every method, but you never pass it explicitly; Python fills it in automatically:

```
rex.bark()      # Python actually calls this as:
Dog.bark(rex)   # bark(self) receives rex as self
```

The "this" Parameter: Go vs Kotlin vs Python

Language	How the current object is referenced
Go	No classes at all — a method has an explicit receiver, e.g. `func (d Dog) Bark() string`, conceptually similar to `self` but attached to the function signature differently, and structs/methods are separate constructs rather than one class.
Kotlin	`this` is implicit — never declared as a parameter, just used directly inside a method when needed, and usually omittable entirely (`name` instead of `this.name`).
Python	`self` must be declared explicitly as the first parameter of every method, and used explicitly (`self.name`) to access instance attributes — nothing is implicit here.

Coming from Kotlin, writing out `self` everywhere feels verbose at first — it's a deliberate Python design choice ("explicit is better than implicit"), not an oversight.

📌 Instance Attributes vs Class Attributes

```
class Dog:
    species = "Canis familiaris" # CLASS attribute — shared by every Dog
def __init__(self, name):
        self.name = name           # INSTANCE attribute — unique per object

rex = Dog("Rex")
fido = Dog("Fido")

print(rex.name, fido.name)      # Rex Fido — different per instance
print(rex.species, fido.species) # Canis familiaris Canis familiaris — same for both
```

An **instance attribute** (set via `self.x = ...`, usually in `__init__`) belongs to one specific object. A **class attribute** (defined directly in the class body) is shared by every instance of that class — change it via the class itself, and every instance sees the update.

⚠ Mutable Class Attributes Are Shared — And Mutating Them Leaks Across Instances

```
class Dog:
    tricks = []   # DANGER: one list, shared by every Dog
def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)   # mutates the SHARED class list

rex = Dog("Rex")
fido = Dog("Fido")
rex.add_trick("sit")

print(fido.tricks)   # ['sit'] — Fido "learned" Rex's trick!
```

This is the same root cause as Course 1's mutable-default-argument gotcha (Chapter 8) — a single mutable object gets shared where independent copies were expected. The fix: give each instance its own list inside `__init__`, with `self.tricks = []`, not as a class attribute.

class / __init__

`class Name:` defines a type; `__init__` runs automatically on creation.

self

Explicit first parameter of every method — refers to the specific object being used.

Instance attributes

`self.x = ...` — unique data per object, usually set in `__init__`.

Class attributes

Defined in the class body — shared by every instance; avoid mutable ones.

💻 Coding Challenges

Challenge 1: Bank Account Class

Write a `BankAccount` class with an `__init__` that takes `owner` and an optional `balance` (default 0), plus a `deposit(amount)` method that adds to the balance. Create an account and deposit twice, printing the final balance.

Goal: Practice defining a class with instance attributes and a method that modifies them.

→ Solution

Challenge 2: Class Attribute Counter

Write a `Robot` class with a class attribute `count = 0` that increments by 1 every time a new `Robot` is created (inside `__init__`). Create three robots, then print the total count.

Goal: Practice a legitimate, safe use of a class attribute — one that's read/incremented as a number, not a shared mutable container.

→ Solution

Challenge 3: Fix the Shared-List Bug

Given the buggy `Dog` class from this chapter's warning box (with `tricks = []` as a class attribute), rewrite it so each dog's `tricks` list is independent, and prove the fix by showing `fido.tricks` stays empty after `rex.add_trick("sit")`.

Goal: Apply the fix for the shared-mutable-class-attribute gotcha directly.

→ Solution

🎯 What's Next

Next chapter: **OOP Deep Dive** — inheritance, polymorphism, and magic/dunder methods like `__str__` and `__eq__`.

🔗 OOP Deep Dive

Chapter 1 covered defining a single class. This chapter covers three ways classes work *together* and with the rest of the language: inheritance (building a class on top of another), polymorphism (different classes responding to the same method call), and magic/dunder methods — the hooks that let your own objects work with `print()`, `==`, and other built-in behavior.

🧬 Inheritance

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

    def speak(self):
        return f"{self.name} makes a sound"
class Dog(Animal):          # Dog inherits from Animal
def __init__(self, name, breed):
        super().__init__(name)   # call Animal's __init__ first
        self.breed = breed

    def speak(self):            # overriding Animal's speak()
return f"{self.name} says Woof!"

rex = Dog("Rex", "Labrador")
print(rex.speak())   # Rex says Woof! — Dog's own version
```

`class Dog(Animal):` makes `Dog` a subclass of `Animal`, inheriting its attributes and methods. `super().__init__(name)` calls the parent class's constructor, so `Dog` doesn't have to duplicate the logic for setting `self.name`.

⚠ Forgetting super().__init__() Silently Skips Parent Setup

If `Dog.__init__` sets `self.breed` but never calls `super().__init__(name)`, the object never gets a `self.name` attribute at all — calling `rex.name` later raises an `AttributeError`, and the failure happens far away from the actual mistake, making it a confusing bug to track down. Whenever a subclass defines its own `__init__`, call `super().__init__(...)` first unless you deliberately want to skip the parent's setup.

🎭 Polymorphism

Different classes can implement the same method name, and calling code can treat them uniformly without checking which specific class it's dealing with:

```
class Cat(Animal):
    def speak(self):
        return f"{self.name} says Meow!"

animals = [Dog("Rex", "Labrador"), Cat("Whiskers")]

for animal in animals:
    print(animal.speak())   # each object uses its OWN speak(), automatically
# Rex says Woof!
# Whiskers says Meow!
```

The loop doesn't need an `if isinstance(animal, Dog)` check anywhere — calling `.speak()` automatically runs whichever version belongs to that object's actual class. This is polymorphism: one interface (`.speak()`), many implementations.

✨ Magic / Dunder Methods

"Dunder" (double underscore) methods let your class hook into Python's built-in behavior. `__init__` is one you already know — here are two more of the most common:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return f"({self.x}, {self.y})"
def __eq__(self, other):
        return self.x == other.x and self.y == other.y

p1 = Point(1, 2)
p2 = Point(1, 2)

print(p1)       # (1, 2) — uses __str__, not "<__main__.Point object at 0x...>"
print(p1 == p2)  # True — uses __eq__, compares VALUES not identity
```

⚠ Without __eq__, == Compares Identity, Not Values

Without a custom `__eq__`, `p1 == p2` would be `False` even with identical `x`/`y` values — the default `==` checks whether both names point to the exact same object in memory, the same identity-vs-equality distinction that matters for strings and numbers too, but is easy to forget once you're writing your own classes.

Similarly, without `__str__`, `print(p1)` would show something unhelpful like `<__main__.Point object at 0x000001A2B3C4D5E6>` — the default representation, which just identifies the object's type and memory address rather than anything about its actual data.

Inheritance & Object Representation: Go vs Kotlin vs Python

Feature	Go	Kotlin	Python
Inheritance	No classes/inheritance — struct embedding approximates it	Classes are `final` by default; must mark `open class` to allow subclassing	Every class is inheritable by default, no keyword needed
Custom string form	Implement the `Stringer` interface's `String()` method	Override `toString()`	Define `__str__`
Custom equality	Manual field-by-field comparison (or reflect.DeepEqual)	`data class` generates `equals()` automatically	Define `__eq__` manually

Kotlin's `open class` requirement is a real, easy-to-forget difference coming from Python — Python never blocks subclassing by default, where Kotlin does unless you opt in.

Inheritance

`class Child(Parent):`, with `super()` to call the parent's methods.

Overriding

Redefining a parent's method in the subclass to change its behavior.

Polymorphism

Calling code uses one method name; each object's own class decides what actually runs.

Dunder methods

`__str__` for `print()`, `__eq__` for `==` — hooks into built-in behavior.

💻 Coding Challenges

Challenge 1: Shape Hierarchy

Write a base class `Shape` with a method `area()` that returns 0, then subclasses `Square(side)` and `Circle(radius)` that each override `area()` with their correct formula. Loop over a list of one of each and print each shape's area.

Goal: Practice inheritance and overriding together, plus polymorphic iteration.

→ Solution

Challenge 2: A Printable Fraction Class

Write a `Fraction` class with `numerator` and `denominator` instance attributes, and a `__str__` method that prints it as `"numerator/denominator"` (e.g. `"3/4"`).

Goal: Practice writing a `__str__` method and seeing it kick in automatically with `print()`.

→ Solution

Challenge 3: Value-Based Equality

Add an `__eq__` method to the `Fraction` class from Challenge 2, so that two `Fraction` objects with the same numerator and denominator compare as equal with `==`. Prove it works by comparing two separately created but equal fractions.

Goal: Practice writing `__eq__` and understand why it's needed for value-based comparison.

→ Solution

🎯 What's Next

Next chapter: **Exception Handling** — `try`/`except`/`else`/`finally`, raising exceptions, and custom exceptions.

🚨 Exception Handling

You've already seen a few exceptions crash a program by accident — a `KeyError` from a missing dict key (Course 1, Chapter 7), a `TypeError` from mutating a tuple. This chapter covers handling them deliberately: `try/except/else/finally`, raising your own exceptions, and defining custom exception types.

🛡️ try/except Basics

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Can't divide by zero!")

# catching multiple, specific exception types:
try:
    value = int(input("Enter a number: "))
except ValueError:
    print("That wasn't a valid number")
except KeyboardInterrupt:
    print("Input cancelled")
```

Code inside `try` runs normally until an exception occurs; the matching `except` block then runs instead of letting the program crash. Multiple `except` blocks let you handle different failure types differently.

⚠ A Bare except: Catches Everything — Including Things You Don't Want

```
try:
    risky_operation()
except:   # catches ALL exceptions, even ones you didn't anticipate
print("something went wrong")
```

A bare `except:` with no exception type swallows everything — including typos that raise `NameError`, programming bugs that raise `TypeError`, and even `KeyboardInterrupt` (Ctrl+C) and `SystemExit`, making the program impossible to stop cleanly. Always catch specific exception types (`except ValueError:`), or at minimum `except Exception:`, which excludes those system-level signals.

🔀 else and finally

```
try:
    value = int("42")
except ValueError:
    print("conversion failed")
else:
    print(f"conversion succeeded: {value}")   # runs only if NO exception occurred
finally:
    print("this always runs")                 # runs no matter what
```

This is the same `else`-means-"no exception fired" idea as the loop `else` clause from Course 1, Chapter 4 — `else` here runs only when the `try` block completed with no exception. `finally` always runs, whether an exception occurred or not — the standard place for cleanup code (closing a file or network connection) that must happen either way.

📢 Raising Exceptions

```
def withdraw(balance, amount):
    if amount > balance:
        raise ValueError("Insufficient funds")
    return balance - amount

try:
    withdraw(100, 150)
except ValueError as e:
    print(f"Error: {e}")   # Error: Insufficient funds
```

`raise` triggers an exception deliberately — `as e` captures the exception object so you can inspect its message with `str(e)` or, as shown, directly in an f-string.

🏷️ Custom Exceptions

```
class InsufficientFundsError(Exception):
    pass
def withdraw(balance, amount):
    if amount > balance:
        raise InsufficientFundsError(f"Cannot withdraw {amount}, balance is only {balance}")
    return balance - amount

try:
    withdraw(100, 150)
except InsufficientFundsError as e:
    print(e)   # Cannot withdraw 150, balance is only 100
```

A custom exception is just a class inheriting from `Exception` (usually with nothing else needed — `pass` is enough) — reusing the exact class/inheritance mechanics from Chapter 2. Naming it specifically (`InsufficientFundsError` rather than a generic `ValueError`) makes both the raising code and the catching code more self-documenting.

Error Handling: Go vs Kotlin vs Python

Language	Approach
Go	No exceptions at all — functions return an explicit `(value, error)` pair, and the caller checks `if err != nil` after every call. Failure is just a normal return value.
Kotlin	Exceptions, like Java — but all unchecked (no `throws` declarations the compiler enforces), so a function's signature alone never tells you what it might throw.
Python	Exceptions, always unchecked, propagating up the call stack until caught by a `try`/`except` or crashing the program.

Go's explicit-error-return style is a genuinely different philosophy from Python's — every Go function that can fail says so directly in its return type, where Python (and Kotlin) require reading the implementation or documentation to know what might be raised.

try / except

Catches specific exception types; avoid a bare `except:`.

else / finally

`else` runs only on success; `finally` always runs, for cleanup.

raise

Triggers an exception deliberately, optionally with a message.

Custom exceptions

A class inheriting from `Exception` — self-documenting failure types.

💻 Coding Challenges

Challenge 1: Safe Division Function

Write a function `safe_divide(a, b)` that returns `a / b`, but catches `ZeroDivisionError` and returns `None` instead of crashing when `b` is 0. Call it with `(10, 2)` and `(10, 0)`, printing both results.

Goal: Practice basic `try`/`except` around a real failure case.

→ Solution

Challenge 2: Validated Age Input

Write a function `set_age(age)` that raises a `ValueError` with the message `"Age cannot be negative"` if `age < 0`, and otherwise returns the age unchanged. Call it with a valid age inside a `try`/`except`, then with `-5`, printing the caught error message.

Goal: Practice `raise`-ing a built-in exception type with a custom message, and catching it with `as e`.

→ Solution

Challenge 3: Custom Exception for Stock Levels

Define a custom exception `OutOfStockError(Exception)`. Write a function `purchase(stock, quantity)` that raises it (with a descriptive message) if `quantity > stock`, and otherwise returns `stock - quantity`. Demonstrate it raising and being caught.

Goal: Practice defining and using a custom exception end to end.

→ Solution

🎯 What's Next

Next chapter: **File I/O** — reading/writing files, context managers (the `with` statement), and `pathlib`.

📁 File I/O

Everything so far has lived only in memory while the program runs. This chapter covers reading and writing real files: `open()` and its modes, the `with` statement (a context manager) that guarantees files get closed properly, and `pathlib` — the modern, object-oriented way to work with file paths.

📖 Opening and Reading Files

```
file = open("notes.txt", "r")   # "r" = read mode (the default)
contents = file.read()          # the entire file, as one string
file.close()                    # must close manually — see the warning below

file = open("notes.txt", "r")
lines = file.readlines()        # a list of strings, one per line
file.close()

file = open("notes.txt", "r")
for line in file:               # iterate line by line — memory-efficient for large files
print(line.strip())             # .strip() removes the trailing newline
file.close()
```

🔒 The with Statement (Context Managers)

```
with open("notes.txt", "r") as file:
    contents = file.read()
# file is automatically closed here, even if an exception occurred inside the block
```

`with` is a **context manager** — it guarantees cleanup code runs when the block exits, whether that's normally or via an exception, the same guarantee `finally` gave you in the last chapter. For files specifically, this is the idiomatic form; plain `open()`/`.close()` is considered old-fashioned in real code.

⚠ Forgetting .close() Leaks a File Handle

If an exception happens between a manual `open()` and its matching `.close()`, the `close()` line is skipped entirely — the file stays open, holding onto a system resource, for as long as the program keeps running. `with` closes the file automatically no matter how the block exits, which is why it's always preferred over manual `open`/`close` pairs.

🖍️ Writing Files

```
with open("log.txt", "w") as file:   # "w" = write mode
    file.write("First line\n")
    file.write("Second line\n")

with open("log.txt", "a") as file:   # "a" = append mode
    file.write("Third line\n")
```

⚠ "w" Mode Erases the File's Existing Contents Immediately

Opening a file in `"w"` mode **truncates it to empty the instant it's opened** — before you've written anything new. Opening an existing file you meant to add to using `"w"` instead of `"a"` silently destroys everything already in it. Double-check which mode you actually want before running write code against a real file.

🗂️ pathlib

`pathlib` represents file paths as objects rather than plain strings, with cross-platform path joining and useful methods built in:

```
from pathlib import Path

data_dir = Path("data")
file_path = data_dir / "notes.txt" # the / operator joins paths — works on Windows AND Linux
print(file_path.exists())     # True/False — no need to open() just to check
print(file_path.name)         # notes.txt
print(file_path.suffix)       # .txt

contents = file_path.read_text()        # reads the whole file — no open()/close() needed
file_path.write_text("new contents")   # writes the whole file in one call
```

`Path`'s `/` operator is a genuinely nice piece of operator overloading — it replaces manually concatenating strings with slashes (which breaks on Windows anyway, since Windows uses backslashes) with something that reads naturally and works everywhere.

Guaranteed Cleanup: Go vs Kotlin vs Python

Language	Mechanism
Go	`defer file.Close()` — scheduled to run when the enclosing function returns, regardless of how it returns (including via an error).
Kotlin	`file.bufferedReader().use { ... }` — the `use` function is Kotlin's own context-manager equivalent, closing the resource automatically at the end of the block.
Python	`with open(...) as file:` — the file closes automatically when the `with` block ends.

All three languages converged on the same underlying idea — guaranteed cleanup tied to a block's exit, not left to manual discipline — just with different syntax and a different name for the concept.

open() modes

`"r"` read, `"w"` write (truncates!), `"a"` append.

with statement

Guarantees the file closes automatically, even if an exception occurs.

Reading

`.read()` (whole file), `.readlines()` (list of lines), or iterate directly.

pathlib

`Path` objects, `/` for joining, `.read_text()`/`.write_text()` shortcuts.

💻 Coding Challenges

Challenge 1: Write Then Read

Using `with` and `"w"` mode, write three lines to a file called `output.txt`. Then, in a separate `with` block using `"r"` mode, read and print each line (with `.strip()` to remove trailing newlines).

Goal: Practice the write-then-read round trip using context managers for both.

→ Solution

Challenge 2: Line Counter

Given a text file with several lines, write a function `count_lines(filename)` that opens it with `with` and returns how many lines it contains, by iterating over the file object directly rather than calling `.readlines()`.

Goal: Practice line-by-line iteration over a file object as a memory-efficient alternative to reading everything into a list.

→ Solution

Challenge 3: pathlib File Check

Using `pathlib`, write code that checks whether a file called `config.txt` exists in a folder called `settings`. If it exists, print its `.suffix`; if not, use `.write_text()` to create it with the contents `"default config"`.

Goal: Practice combining `Path`'s `/` operator, `.exists()`, and `.write_text()` in one realistic check-then-act flow.

→ Solution

🎯 What's Next

Next chapter: **Comprehensions & Generators** — list/dict/set comprehensions in depth, generator functions, and `yield`.

⚙️ Comprehensions & Generators

Course 1 gave you a first look at list comprehensions (Chapter 6) and dict comprehensions (Chapter 7). This chapter goes deeper: set comprehensions, nested comprehensions, and — the real payoff — generator expressions and generator functions with `yield`, which produce values lazily instead of building an entire collection in memory up front.

🎯 Set Comprehensions

The same `{expr for item in iterable}` shape works for sets too, just with automatic deduplication:

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

lengths = {len(word) for word in words}
print(lengths)   # {5, 6} — duplicate 5s from "apple" and "apple" collapse to one
```

🪆 Nested Comprehensions

```
matrix = [[1, 2, 3], [4, 5, 6]]

# flatten a list of lists into one list:
flat = [n for row in matrix for n in row]
print(flat)  # [1, 2, 3, 4, 5, 6]
```

Reading order matters: the `for` clauses read left to right exactly like nested `for` loops would — `for row in matrix` is the outer loop, `for n in row` is the inner one. Written as loops, this would be:

```
flat = []
for row in matrix:
    for n in row:
        flat.append(n)
```

🌊 Generator Expressions

A generator expression looks almost identical to a list comprehension, but with parentheses instead of square brackets — and it computes values one at a time, on demand, rather than building the whole list up front:

```
squares_list = [n ** 2 for n in range(1000000)]   # builds all 1,000,000 values NOW, in memory
squares_gen  = (n ** 2 for n in range(1000000))   # computes each value only when asked
print(type(squares_gen))   # <class 'generator'>
print(next(squares_gen))   # 0 — the first value, computed just now
print(next(squares_gen))   # 1 — the second value
```

For a million items, the list comprehension allocates memory for all million numbers immediately; the generator expression holds almost no memory at all — just enough to know where it left off, computing the next value only when `next()` (or a `for` loop) asks for it.

⏸ Generator Functions and yield

```
def count_up_to(n):
    i = 1
while i <= n:
        yield i    # pauses here, hands back i, remembers where it was
        i += 1
for num in count_up_to(5):
    print(num)
# 1 2 3 4 5
```

Any function containing `yield` becomes a **generator function** — calling it doesn't run the body immediately, it returns a generator object. Each call to `next()` (which a `for` loop does automatically) resumes the function right where it last paused, runs until the next `yield`, and hands back that value — the function's local state (like `i` here) is preserved between calls, something a regular function can't do at all.

⚠ A Generator Is Exhausted After One Full Pass

```
gen = count_up_to(3)
print(list(gen))   # [1, 2, 3]
print(list(gen))   # [] — empty! the generator is already exhausted
```

Unlike a list, a generator can only be iterated once — once every value has been produced, it's done, and iterating it again yields nothing. If you need to loop over the same values twice, either use a list/list comprehension instead, or call the generator function again to get a fresh generator object.

Lazy Iteration: Go vs Kotlin vs Python

Language	Approach
Go	No generator syntax — lazy, on-demand value production is typically built manually with goroutines and channels, a heavier-weight tool for the same underlying idea.
Kotlin	`sequence { yield(value) }` — genuinely close to Python's `yield`, producing a lazy `Sequence` the same way Python's generator functions produce a lazy generator.
Python	`yield` inside any function, or a generator expression `(x for x in ...)`.

Kotlin's `sequence { yield(...) }` is one of the closer direct parallels to a Python feature covered so far in this course — the underlying "pause and resume, producing values lazily" mechanism is essentially the same idea in different syntax.

Set comprehension

`{expr for item in iterable}` — same shape as list/dict comprehensions.

Nested comprehension

Multiple `for` clauses read left to right, like nested loops.

Generator expression

`(expr for item in iterable)` — lazy, memory-efficient, one value at a time.

yield

Pauses a function, hands back a value, and resumes exactly there on the next call.

💻 Coding Challenges

Challenge 1: Unique Word Lengths

Given `sentence = "the quick brown fox jumps over the lazy dog"`, use a set comprehension to find the set of unique word lengths appearing in the sentence.

Goal: Practice a set comprehension combined with `.split()` from Course 1.

→ Solution

Challenge 2: Flatten and Filter

Given `matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]`, use a single nested comprehension (with an `if` condition) to produce a flat list of only the even numbers: `[2, 4, 6, 8]`.

Goal: Practice combining nested comprehensions with a filtering condition in one expression.

→ Solution

Challenge 3: A Fibonacci Generator

Write a generator function `fibonacci(n)` that yields the first `n` numbers of the Fibonacci sequence (0, 1, 1, 2, 3, 5, ...), one at a time. Use a `for` loop to print all values from `fibonacci(8)`.

Goal: Practice writing a generator function with `yield` that maintains state across calls.

→ Solution

🎯 What's Next

Next chapter: **Decorators** — function decorators, `@property`, and `functools` (`wraps`, `lru_cache`).

🎀 Decorators

A decorator wraps a function to add behavior around it — logging, timing, caching, access control — without changing the function's own code. This chapter covers how decorators actually work under the hood, `functools.wraps` (a fix for a subtle side effect), `@property` for attribute-like method access, and `functools.lru_cache` for automatic memoization.

🔧 Functions Are Objects

Decorators only make sense once you know functions in Python are ordinary objects — they can be assigned to variables, passed as arguments, and returned from other functions:

```
def say_hello():
    return "Hello!"

greeting = say_hello   # no parentheses — assigning the FUNCTION itself, not calling it
print(greeting())    # Hello! — greeting is just another name for the same function
def run_twice(func):    # a function that takes ANOTHER function as an argument
    func()
    func()

run_twice(say_hello)   # calls say_hello() twice
```

🎁 Writing a Basic Decorator

```
def shout(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        return result.upper()
    return wrapper

@shout
def greet(name):
    return f"hello, {name}"
print(greet("Ada"))   # HELLO, ADA
```

`@shout` above `def greet` is syntax sugar for `greet = shout(greet)` — `greet` now actually refers to `wrapper`, which calls the original `greet` logic internally (as `func`) and transforms its result before returning it.

⚠ Always Use *args, **kwargs in the Wrapper

If `wrapper` is defined as `def wrapper():` with no parameters, decorating `greet(name)` breaks entirely — calling `greet("Ada")` actually calls `wrapper("Ada")`, which doesn't accept any arguments. Using `*args, **kwargs` (Course 1, Chapter 8) lets the wrapper forward whatever arguments it received straight through to `func`, so the decorator works on any function regardless of its parameter list.

🏷️ functools.wraps

```
print(greet.__name__)   # wrapper — NOT "greet"! Metadata got lost.
from functools import wraps

def shout(func):
    @wraps(func)        # preserves func's __name__, docstring, etc.
def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        return result.upper()
    return wrapper

@shout
def greet(name):
    return f"hello, {name}"
print(greet.__name__)   # greet — fixed!
```

Without `@wraps(func)`, the decorated function's identity (its `__name__`, docstring, and other metadata) silently becomes `wrapper`'s instead of the original function's — confusing for debugging, introspection, and tools that rely on that metadata. `@wraps(func)` from `functools` is standard practice in every real decorator.

📐 @property

```
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
def area(self):
        return 3.14159 * self._radius ** 2

c = Circle(4)
print(c.area)   # 50.26544 — accessed like an ATTRIBUTE, no parentheses, but computed live
```

`@property` makes a method accessible as if it were a plain attribute — `c.area`, not `c.area()` — while still running real code every time it's read. This is useful for a value that's always derived from other attributes (like `area` from `radius`) rather than stored directly.

💾 functools.lru_cache

```python
from functools import lru_cache

@lru_cache
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

print(fibonacci(30))   # fast — each fibonacci(n) is only ever actually computed once
```

`@lru_cache` automatically remembers previous return values for previous arguments, so calling `fibonacci(30)` a second time (or any recursive call the first computation already made) returns the cached result instantly instead of recomputing it. This turns the naive recursive Fibonacci implementation — normally exponentially slow — into something fast, with a single decorator line and no change to the function's actual logic.

Wrapping Behavior: Go vs Kotlin vs Python

Language	Approach
Go	No decorator syntax at all — the same effect is achieved manually with higher-order functions (a function that takes and returns a `func`) or the "middleware" pattern common in HTTP handlers.
Kotlin	No direct decorator syntax either — achieved with higher-order functions and lambdas, or with annotations processed at compile/build time for a different (metadata-driven) purpose.
Python	First-class `@decorator` syntax, built directly into the language.

Python's `@decorator` syntax is a genuine convenience neither Go nor Kotlin has natively — both can express the same underlying higher-order-function idea, just without the dedicated syntax sugar.

@decorator

Sugar for `func = decorator(func)` — wraps a function with extra behavior.

*args, **kwargs

Required in the wrapper so it can forward any function's arguments through.

@property

Makes a method readable like a plain attribute, computed live.

@lru_cache

Automatic memoization — remembers results for previously-seen arguments.

💻 Coding Challenges

Challenge 1: A Timing Decorator

Write a decorator `@timer` that prints how long the decorated function took to run (use the `time` module's `time.time()` before and after calling the function), then returns the function's original result unchanged. Apply it to any simple function.

Goal: Practice writing a decorator that measures something without altering the wrapped function's return value.

→ Solution

Challenge 2: Rectangle with @property

Write a `Rectangle` class with `width` and `height` instance attributes, plus an `@property` method `area` that returns `width * height`. Create an instance and print its `.area`.

Goal: Practice defining and using a computed `@property`.

→ Solution

Challenge 3: Cached Fibonacci, Verified

Using `@lru_cache`, write the recursive `fibonacci(n)` function from this chapter, then call `fibonacci(35)` and print the result — something the same function without `@lru_cache` would take noticeably longer to compute.

Goal: See `@lru_cache`'s real performance impact firsthand, not just read about it.

→ Solution

🎯 What's Next

Next chapter: **Working with Data** — JSON, CSV, and the `datetime` module.

🗂️ Working with Data

Real programs constantly move data in and out of a program — to a file, an API, a spreadsheet. This chapter covers the three tools you'll reach for most: the `json` module for JSON data, the `csv` module for spreadsheet-style data, and a deeper look at `datetime`, which you saw briefly in Course 1, Chapter 9.

📋 JSON

```
import json

person = {"name": "Ada", "age": 28, "active": True}

json_string = json.dumps(person)    # Python dict → JSON string
print(json_string)
# {"name": "Ada", "age": 28, "active": true}

parsed = json.loads(json_string)    # JSON string → Python dict
print(parsed["name"])               # Ada
# reading/writing JSON files directly:
with open("person.json", "w") as f:
    json.dump(person, f)            # note: dump (file), not dumps (string)
with open("person.json", "r") as f:
    loaded = json.load(f)
```

JSON's types map onto Python's fairly directly: JSON objects become dicts, arrays become lists, `true`/`false` become `True`/`False`, and `null` becomes `None`. `dumps`/`loads` (with an "s", for "string") convert to and from a Python string; `dump`/`load` (no "s") read and write a file directly.

⚠ json.dumps() Can't Serialize Every Python Type

```
import datetime
data = {"created": datetime.date.today()}
json.dumps(data)  # TypeError: Object of type date is not JSON serializable
```

JSON only understands strings, numbers, booleans, `null`, objects, and arrays — a Python `date` object (or any other custom type) has no direct JSON equivalent. The fix is to convert it to a string first (e.g. `str(datetime.date.today())` or `.isoformat()`) before serializing, or pass a custom `default=` function to `json.dumps()` that tells it how to convert non-standard types.

📊 CSV

```
import csv

# writing rows as plain lists:
with open("scores.csv", "w", newline="") as f:
    writer = csv.writer(f)
    writer.writerow(["name", "score"])
    writer.writerow(["Alice", 90])
    writer.writerow(["Bob", 85])

# reading rows back as plain lists:
with open("scores.csv", "r", newline="") as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)  # ['name', 'score'], ['Alice', '90'], ['Bob', '85']
# DictReader/DictWriter — rows as dicts, keyed by the header row:
with open("scores.csv", "r", newline="") as f:
    reader = csv.DictReader(f)
    for row in reader:
        print(row["name"], row["score"])  # Alice 90, Bob 85
```

`DictReader`/`DictWriter` use the file's header row as dictionary keys — usually more convenient than `reader`/`writer`'s plain lists, since you access columns by name (`row["score"]`) instead of by position (`row[1]`), the same readability trade-off dicts always have over lists.

The datetime Module, Deeper

```
import datetime

now = datetime.datetime.now()        # current date AND time (datetime.date.today() is date-only)
# formatting a datetime as a string:
formatted = now.strftime("%Y-%m-%d %H:%M")
print(formatted)   # 2026-07-09 14:32
# parsing a string back into a datetime:
parsed = datetime.datetime.strptime("2026-01-01", "%Y-%m-%d")

# timedelta arithmetic:
one_week_later = now + datetime.timedelta(days=7)
print((one_week_later - now).days)   # 7
```

`strftime` ("string format time") turns a `datetime` into a string using a format code like `%Y-%m-%d`; `strptime` ("string parse time") does the reverse, turning a string into a `datetime` given the format it's in. Adding/subtracting a `timedelta` — as briefly seen with subtraction in Course 1's own challenge solutions — handles date arithmetic without manually accounting for month lengths or leap years.

Structured Data Serialization: Go vs Kotlin vs Python

Language	Approach
Go	`encoding/json` maps JSON onto real, statically-typed `struct` fields using `json:"field_name"` struct tags — the shape is declared and checked at compile time.
Kotlin	`kotlinx.serialization` (or Gson/Jackson) similarly maps JSON onto typed `data class` fields, checked by the compiler.
Python	`json.loads()` returns a plain, untyped `dict` — no schema is declared or checked; a missing or misspelled key only fails at runtime, as a `KeyError`.

This is a real trade-off worth noticing coming from Go or Kotlin: Python's approach is faster to write and more flexible, but loses the compile-time guarantee that the JSON actually matches the shape your code expects.

json.dumps/loads

Python dict/list ↔ JSON string; `dump`/`load` for files directly.

csv.reader/writer

Rows as plain lists; `DictReader`/`DictWriter` for rows as dicts by column name.

strftime / strptime

Format a `datetime` to a string, or parse a string into a `datetime`.

timedelta

Date/time arithmetic that correctly handles month lengths and leap years.

💻 Coding Challenges

Challenge 1: Round-Trip a Dict Through JSON

Given a dict representing a small product (`name`, `price`, `in_stock`), convert it to a JSON string with `json.dumps()`, then parse that string back into a Python dict with `json.loads()`, and confirm the round trip produced an equal dict.

Goal: Practice the full `dumps` → `loads` round trip and confirm no data was lost.

→ Solution

Challenge 2: CSV Average Score

Given a CSV file with columns `name` and `score`, use `csv.DictReader` to read every row and calculate the average score across all students (remembering that CSV values are read as strings, so each score needs converting to a number first).

Goal: Practice `DictReader` combined with numeric type conversion, a common real-world CSV gotcha.

→ Solution

Challenge 3: Days Until a Parsed Deadline

Given a deadline as a string, `"2026-12-25"`, use `datetime.strptime()` to parse it into a `date`, then calculate and print how many days remain between today and that date.

Goal: Practice `strptime` parsing combined with date subtraction.

→ Solution

🎯 What's Next

Next chapter: **Testing with pytest** — assertions, fixtures, `parametrize`, and a contrast with the built-in `unittest` module.

🧪 Testing with pytest

The final chapter of Python Intermediate: writing automated tests with `pytest`, the de facto standard testing tool in the Python ecosystem. This covers plain `assert`-based tests, fixtures for shared setup, `parametrize` for running one test against many inputs, and how `pytest` compares to Python's built-in `unittest` module.

✅ Why Test? A Quick assert Primer

```
assert 2 + 2 == 4 # passes silently — nothing happens
assert 2 + 2 == 5, "math is broken" # raises AssertionError: math is broken
```

`assert` is a plain Python keyword — if the condition is true, nothing happens; if it's false, it raises an `AssertionError` (optionally with a message). `pytest` builds its entire testing style around this one familiar keyword, rather than introducing a whole new assertion API.

🔬 Writing Tests with pytest

```
# test_math.py
def add(a, b):
    return a + b

def test_add_positive_numbers():
    assert add(2, 3) == 5
def test_add_negative_numbers():
    assert add(-1, -1) == -2

# run from the command line:
pytest test_math.py

# output:
# test_math.py::test_add_positive_numbers PASSED
# test_math.py::test_add_negative_numbers PASSED
```

`pytest` auto-discovers test functions by naming convention alone: any file named `test_*.py` (or `*_test.py`), containing functions named `test_*`, is picked up automatically — no manual test registration, and no base class to inherit from, unlike `unittest` (see below).

🧩 Fixtures

```
import pytest

@pytest.fixture
def sample_list():
    return [1, 2, 3]

def test_sum(sample_list):        # pytest injects the fixture by matching the parameter name
assert sum(sample_list) == 6
def test_length(sample_list):
    assert len(sample_list) == 3
```

A **fixture** is a function decorated with `@pytest.fixture` that provides shared setup for tests — any test function that takes a parameter matching the fixture's name automatically receives its return value. This is `pytest`'s own dependency-injection style, avoiding the repeated setup code every test would otherwise need.

```
@pytest.fixture
def temp_file():
    f = open("temp.txt", "w")
    yield f      # the test runs here, using f
    f.close()   # cleanup, guaranteed to run after the test finishes
```

A fixture that uses `yield` instead of `return` gets a setup/teardown split — code before `yield` runs before the test, code after `yield` runs after, reusing the exact `yield`-pauses-and-resumes mechanic from Chapter 5's generator functions.

🔁 parametrize

```
@pytest.mark.parametrize("a, b, expected", [
  (2, 3, 5),
  (-1, 1, 0),
  (0, 0, 0),
])
def test_add(a, b, expected):
  assert add(a, b) == expected
```

`@pytest.mark.parametrize` runs the same test function once per tuple of inputs, reported as three separate, individually pass/fail test results — instead of writing three nearly-identical test functions by hand, or looping inside a single test where one failure would hide the others.

⚠ Never assert Equality on Floats Directly

```
assert 0.1 + 0.2 == 0.3 # fails! 0.1 + 0.2 is actually 0.30000000000000004
import pytest
assert 0.1 + 0.2 == pytest.approx(0.3)  # passes — allows for tiny floating-point error
```

Floating-point numbers can't represent most decimal values exactly, so arithmetic that looks correct on paper often produces a result that's off by a tiny amount — this isn't a Python bug, it's how binary floating-point works in every language. `pytest.approx()` compares with a small allowed tolerance instead of exact equality, exactly for this situation.

pytest vs unittest

	unittest (standard library)	pytest (third-party)
Test structure	Class-based, inheriting from `TestCase`	Plain functions — no class or inheritance required
Assertions	`self.assertEqual(a, b)`, `self.assertTrue(x)`, a whole API of methods to remember	Plain `assert` — one keyword covers everything
Setup/teardown	`setUp()`/`tearDown()` methods	Fixtures with `@pytest.fixture`, more flexible and reusable
Multiple inputs	Manual loops or repeated methods	`@pytest.mark.parametrize`

`unittest` ships with Python itself, so it needs no installation — but `pytest` (a `pip install` away, per Course 1, Chapter 9) is the de facto standard in real-world Python projects for a reason: noticeably less boilerplate per test, and both fixtures and `parametrize` have no clean `unittest` equivalent.

assert

Plain Python keyword — raises `AssertionError` if the condition is false.

Test discovery

`test_*.py` files, `test_*` functions — found automatically, no registration.

Fixtures

`@pytest.fixture` — shared setup, injected by matching parameter name.

parametrize

Runs one test function against many input/expected-output pairs.

💻 Coding Challenges

Challenge 1: Test a String Function

Write a function `is_palindrome(s)` (reusing the logic from Course 1, Chapter 5's palindrome challenge), then write two `pytest` test functions for it: one asserting `is_palindrome("racecar")` is `True`, one asserting `is_palindrome("python")` is `False`.

Goal: Practice writing basic `pytest` test functions with plain `assert`.

→ Solution

Challenge 2: A Fixture for Shared Test Data

Write a `@pytest.fixture` called `sample_scores` that returns the list `[85, 90, 78, 92]`. Then write two tests that both use it — one asserting the average is 86.25, one asserting the max is 92.

Goal: Practice sharing setup data across multiple tests using a fixture instead of repeating the list in every test.

→ Solution

Challenge 3: Parametrized FizzBuzz Test

Write a `fizzbuzz(n)` function (reusing the logic idea from Course 1, Chapter 3's FizzBuzz-style challenge, but returning a string instead of printing), then use `@pytest.mark.parametrize` to test it against at least 4 different inputs and their expected outputs in a single test function.

Goal: Practice `parametrize` to cover multiple cases without writing a separate test function for each.

→ Solution

🎓 Course Complete!

That's all 8 chapters of **Python Intermediate** — OOP, inheritance and dunder methods, exception handling, file I/O, comprehensions and generators, decorators, working with data, and testing with pytest. Next up: **Python Advanced** (`py3`), starting with Advanced OOP.