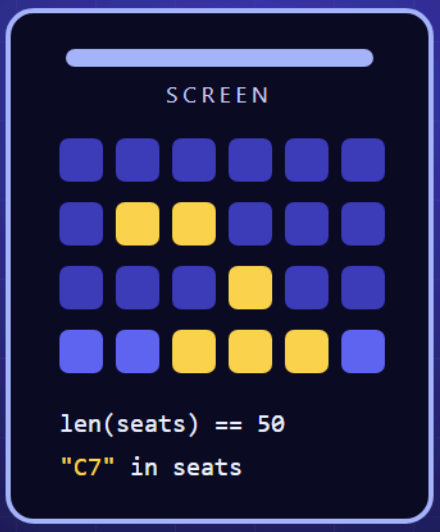
```python
@dataclass(frozen=True)
class Seat:
    row: str
    number: int

class SeatMap:
    def __len__(self): ...
    def __iter__(self): ...
    def __getitem__(self, label): ...
```




```python
def read_rows(path):
    with path.open(encoding="utf-8") as f:
        yield from csv.DictReader(f)

total = sum(t.price
    for t in box_office)
```




```python
@retry(times=3) match cmd.split(): case ["sell", sid, *seats]:
```

INTERMEDIATE PYTHON, DONE RIGHT

Pythonic Python

Write code like an experienced Pythonista — idioms, the data model, and beyond

Comprehensions · Unpacking · Dunder Methods · Iterators
Generators · Decorators · Dataclasses · Enums · Protocols

IDIOMS · DATA MODEL · ITERATION · FUNCTIONS · TYPES

THENEXBOOKS

Pythonic Python

Intermediate Idioms, the Data Model, Iterators, Decorators, and Writing Code Like an Experienced Pythonista

From "it works" to code that reads like Python — one real code base, refactored chapter by chapter

First Edition

THENEXBOOKS

Pythonic Python
Intermediate Idioms, the Data Model, Iterators, Decorators, and Writing Code Like an Experienced Pythonista

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Python and its standard library evolve with every release; behavior, error messages, and tool output can differ between versions. The examples were run on Python 3.12; consult the official Python documentation for the version you use.

"Python" and the Python logos are trademarks or registered trademarks of the Python Software Foundation. Windows is a trademark of the Microsoft group of companies; macOS is a trademark of Apple Inc.; Linux is a registered trademark of Linus Torvalds. pytest, mypy, ruff, pydantic, attrs, and other projects named in this book are the property of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by the Python Software Foundation.

Mapleridge Cinema, the independent cinema used as the running example in this book, is fictional, as are its staff, films, screenings, tickets, and all sample data. Any resemblance to real cinemas, films, or people is coincidental.

Preface

There is a moment every Python developer recognizes. Your code works. The tests pass. And then someone with more Python experience reads it and rewrites twenty lines as four — a comprehension here, a `with` block there, a class that suddenly works with `len()`, `for`, and `sorted()` as if it had always been part of the language. Nothing they used was secret. They simply knew how Python is designed to be used.

That knowledge has a name in the Python community: writing *Pythonic* code. It is not about clever tricks or squeezing logic into one line. It is about working *with* the language — its data model, its protocols, its standard library — so that your code is shorter, clearer, harder to get wrong, and instantly familiar to every other Python developer who reads it.

This book teaches that fluency. We assume you already know Python's basics, and we go straight to the ideas that separate a beginner's Python from an experienced developer's: how names and objects really work, how special methods let your classes plug into the language, how iteration and generators keep programs lazy and memory-friendly, how functions, closures, and decorators compose behavior, and how dataclasses, enums, pattern matching, protocols, and well-designed exceptions make types that are a pleasure to use.

Who This Book Is For

- **Developers who know the basics** — variables, loops, functions, lists, dicts, simple classes, `try` / `except` — and want to write Python that looks like it was written by an expert.
- **Programmers coming from Java, C#, JavaScript, or PHP** who write correct Python that still "feels like another language", and want to learn the idioms instead of translating habits.
- **Self-taught and bootcamp developers** who can build things but want to understand *why* experienced developers reach for `enumerate`, `dict.get`, generators, or `functools`.
- **Team members who review Python code** and want a clear vocabulary and checklist for explaining what "more Pythonic" actually means.

This is not a first Python book: we do not explain what a loop or a function is. It is also not a book about object-oriented design theory, web frameworks, or data science. It is about the core language, used well.

What You Will Learn

- What "Pythonic" really means, how names, references, and mutability work, and how to choose and use lists, tuples, dicts, and sets well.
- Comprehensions, generator expressions, unpacking, slicing, and the small sequence idioms that make code read naturally.

- The Python data model: dunder methods for representation, equality, hashing, ordering, containers, operators, and attribute access — including properties, descriptors, and `__slots__`.
- Iterators, generators, `yield from`, lazy pipelines, the `itertools` module, and context managers with `contextlib`.
- Functions as objects: parameters and signatures, closures, decorators from first principles to decorator factories and typed decorators, and the most useful tools in `functools`.
- Pythonic types: dataclasses, enums, structural pattern matching, the `collections` module, abstract base classes, and protocols.
- Exceptions done right, modules and packages, string formatting and Unicode, and a complete, test-protected refactoring of a real un-Pythonic script.

How This Book Is Organized

Part I — Thinking in Python (Chapters 1–5) defines Pythonic code and covers names and objects, collections, comprehensions, and unpacking and slicing. **Part II — The Data Model** (Chapters 6–10) shows how special methods make your objects behave like built-ins: representation, equality and hashing, container protocols, operator overloading, and attribute access. **Part III — Iteration and Resources** (Chapters 11–14) covers the iteration protocol, generators, `itertools`, and context managers. **Part IV — Functions as Objects** (Chapters 15–19) covers callables and signatures, closures, decorators, and `functools`. **Part V — Designing Pythonic Types** (Chapters 20–24) covers dataclasses, enums, pattern matching, the `collections` module, and abstract base classes and protocols. **Part VI — Pythonic Code in Practice** (Chapters 25–28) covers exceptions, modules and imports, text and formatting, and a full refactoring case study. The appendices give you an idioms cheat sheet, reference tables, a code review checklist, and a glossary.

Each chapter includes runnable examples with their real output, Tips, Notes, and Warnings, a **Try It Yourself** section, and a **Key Takeaways** box for quick review.

The Running Example: Mapleridge Cinema

Throughout the book you maintain the box-office code of **Mapleridge Cinema**, a fictional independent cinema with three screens — Maple, Birch, and Cedar — six films, ten screenings over one weekend, forty tickets, and a busy concession stand. The code base, a small Python package called `mapleridge`, models films, seats and seat maps, screenings, ticket pricing with matinee discounts and premium recliner seats, a box office that sells tickets, and the reports the manager reads every night.

Each chapter improves or adds one piece: a `Money` type that behaves like a number, a seat map that works with `len()`, `in`, and `for`, generators that read data lazily, decorators that log and retry, dataclasses and enums for the domain, pattern matching for box-office commands, and an exception hierarchy that tells callers exactly what went wrong. In the final chapter we take the cinema's original

report script — correct but thoroughly un-Pythonic — and refactor it step by step into clean, idiomatic code, with a test proving the output never changes. The sample data is small enough to check by hand, and every output in this book was produced by running the code.

Versions and Conventions

Examples target **Python 3.12** and run unchanged on newer versions. Where a feature appeared in a recent release, the text says which version introduced it; features of the newest releases (3.13 and later) are mentioned briefly and labeled as such. Commands are shown for Windows as `py -3.12 ...`; on macOS and Linux, type `python3` instead. Tools such as pytest, mypy, and ruff are run as modules (`py -3.12 -m pytest`) so they work even when their scripts are not on your PATH. Output is shown with US English conventions — dates such as *Saturday, March 14, 2026* and amounts such as `$12.50`.

> **TIP**
> A recommended practice or an idiom worth adopting.

> **NOTE**
> Background information, version differences, or a useful aside.

> **WARNING**
> A common mistake, a subtle bug, or behavior that surprises even experienced developers.

Code appears in `monospace` blocks with the file name above them, such as `mapleridge/money.py`; what the code prints appears in a separate block labeled **Output**. Long lines are wrapped to fit the page. Timing and memory-address output varies from machine to machine, and the text says so where it matters.

Contents

Preface

PART I — THINKING IN PYTHON

1 What "Pythonic" Means: Idioms, the Zen, and the Mapleridge Code Base

2 Names, Objects, and References: Identity, Mutability, and Copying

3 Collections Used Well: Lists, Tuples, Dicts, and Sets

4 Comprehensions and Generator Expressions

5 Unpacking, Slicing, and Sequence Idioms

PART II — THE DATA MODEL

6 The Python Data Model: How Dunder Methods Make Objects Feel Built-In

7 Representation, Equality, Hashing, and Ordering

8 Container Protocols: Sized, Iterable, Subscriptable, and Truthy Objects

9 Operator Overloading: Building the Money Type

10 Attribute Access: Properties, Descriptors, and __slots__

PART III — ITERATION AND RESOURCES

11 Iterators and the Iteration Protocol

12 Generators: yield, Lazy Pipelines, and yield from

13 itertools and Lazy Data Processing

14 Context Managers: with, contextlib, and Resource Safety

PART IV — FUNCTIONS AS OBJECTS

15 Functions as First-Class Objects: Callables, Parameters, and Signatures

16 Closures, Scope, and nonlocal

17 Decorators from First Principles

18 Advanced Decorators: Arguments, Classes, Stacking, and Typing

19 functools in Practice: partial, Caching, singledispatch, and reduce

PART V — DESIGNING PYTHONIC TYPES

20 Dataclasses Done Right

21 Enums and Named Constants

22 Structural Pattern Matching with match and case

23 The collections Module: Counter, defaultdict, deque, namedtuple, and ChainMap

24 Abstract Base Classes and Protocols: Duck Typing with Guarantees

PART VI — PYTHONIC CODE IN PRACTICE

25 Exceptions Done Right: Hierarchies, EAFP, Chaining, and Exception Groups

26 Modules, Packages, and Imports

27 Text and String Formatting: f-Strings, Format Specs, and Unicode

28 Refactoring Un-Pythonic Code: The Mapleridge Report Case Study

APPENDICES

A Pythonic Idioms Cheat Sheet

B Templates and Reference

C Pythonic Code Review Checklist

D Glossary

PART I

Thinking in Python

What "Pythonic" really means and why it matters, how names, objects, and references work, choosing and using the built-in collections well, comprehensions and generator expressions, and the unpacking and slicing idioms that make everyday code read naturally.

1 · What "Pythonic" Means

2 · Names, Objects, and References

3 · Collections Used Well

4 · Comprehensions and Generator Expressions

5 · Unpacking, Slicing, and Sequence Idioms

CHAPTER 1 · PART I — THINKING IN PYTHON

What "Pythonic" Means: Idioms, the Zen, and the Mapleridge Code Base

You already write Python that works. This chapter is about writing Python that reads like Python: we define what "Pythonic" really means, meet the cinema whose code you are about to inherit, run its tools for the first time, and do a first small before-and-after rewrite so you can feel the difference.

Welcome to Mapleridge Cinema

Mapleridge Cinema is a small independent cinema with three screens named Maple, Birch, and Cedar. It shows a handful of films each weekend, sells tickets at the box office and online, and sells popcorn, soda, and nachos at the concession stand. Rosa Delgado runs the place, Theo Park builds the weekly schedule and runs the projectors, and Amara Okoye leads the box office. Until recently, all of the cinema's software was written by Sam Whitfield, a part-time developer who taught himself Python on weekends.

Sam has moved on, and you are the new developer. On your first day you find two things in the repository. The first is `legacy/legacy_report.py`, the script Sam wrote years ago to print the daily sales summary that Rosa reads every morning. It works, and nobody wants it to stop working, but it is hard to read and harder to change. The second is the start of a proper package called `mapleridge`, which models films, screens, seats, screenings, tickets, and concessions as real Python objects. That package is the code base we will study, explain, and improve throughout this book.

Every chapter takes one piece of the cinema and makes it more idiomatic: the seat map learns to behave like a built-in container, money becomes a proper type, CSV files are read through lazy generators, reports use format specifications instead of hand-counted spaces, and so on. In the final chapter we come back to Sam's original script and refactor it, step by step, into the package, with a test proving the output never changes. By then you will recognize every move we make, because you will have seen each idiom in isolation first.

This book assumes you already know Python basics: variables, loops, functions, lists and dicts, simple classes, exceptions, and imports. We will not explain what a list is. Instead we will ask better questions about lists: when a tuple is the better choice, why `in` on a set is fast and on a list is slow, and what actually happens when you write `b = a`. If you came to Python from Java, C#, JavaScript, or PHP, you may recognize some of Sam's habits in your own code. That is fine: fluency is learned fastest by comparing a "before" with an "after".

What We Mean by "Pythonic"

"Pythonic" is a word experienced Python developers use all the time, usually without defining it. When someone says a piece of code is "not very Pythonic", they rarely mean it is wrong. They mean it fights the language instead of using it. For this book we use a practical definition with three parts. Pythonic code is:

1. **Readable first.** It says what it means in the vocabulary of the problem. A reader can tell what a line does without simulating index arithmetic in their head.
2. **Built on the language's protocols and built-ins.** It iterates with `for` over the thing itself, asks `len()` for a length, uses `with` for resources, and lets `sum()`, `sorted()`, `any()`, and friends do the work. Its own classes plug into those same protocols, so they feel built-in too.
3. **A good citizen of the ecosystem.** It follows PEP 8 so that every Python developer can read it, it passes the standard tools (a linter, a type checker, a test runner), and it uses the standard library before reinventing it.

Notice what is missing from that list: short, clever, or fast. Pythonic code is often shorter than the alternative, but shortness is a side effect of saying things directly, not a goal. Here is a deliberately bad example. Both versions compute the total ticket revenue for the whole weekend from the sample data.

demo_clever.py

```
from mapleridge import load_cinema

cinema = load_cinema()

# "Clever": one line, three ideas, zero names
print((lambda b: __import__("functools").reduce(lambda a, t: a + t.price.cents, b, 0))(
    cinema.box_office) / 100)

# Pythonic: say what you mean
revenue = sum(ticket.price for ticket in cinema.box_office)
print(revenue)
```

Output

```
408.5
$408.50
```

The first version is a puzzle. It imports a module inside an expression, folds with a lambda, works in raw cents, and converts to a float at the end, which is how it ends up printing `408.5` instead of a dollar amount. The second version reads almost like English: the revenue is the sum of the ticket prices. It works because the cinema's `Money` type cooperates with `sum()`, a design we build in Chapter 9, and because the box office can be iterated like a list, which we explain in Chapter 8. That is the pattern of the whole book: when your objects speak Python's protocols, the code that uses them becomes plain.

> **TIP**
>
> When you are unsure whether something is Pythonic, read the line out loud. If the spoken version sounds like a description of the problem ("the sum of the ticket prices"), you are probably on the right track. If it sounds like a description of the machine ("loop i from zero to length minus one"), look for the idiom.

The Zen of Python

Python ships with a short poem of design principles called the Zen of Python, documented as PEP 20. It is in the standard library, so you can read it at any time:

Terminal

```
py -3.12 -c "import this"
```

Output

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
...
```

The full text has nineteen aphorisms; run the command yourself and read them all, it takes less than a minute. We will not reprint the whole poem, but a few short phrases come up again and again in this book, because they explain *why* an idiom wins:

- **"Readability counts."** Code is read far more often than it is written. Sam's script is read every time something goes wrong; it was written once.
- **"Explicit is better than implicit."** Pass `encoding="utf-8"` to `open()` rather than relying on whatever the operating system picks. Name your constants (`MATINEE_CUTOFF`) instead of hiding the number 17 inside a comparison.
- **"Errors should never pass silently."** A bare `except:` that prints "something went wrong" breaks this rule, and we will fix exactly that in a moment.
- **"There should be one obvious way to do it."** Python often has several ways to do something, but the community converges on one. Learning Python fluently is largely learning which way that is.
- **"Flat is better than nested."** Deeply nested loops and conditionals are a smell. Comprehensions, early returns, and helper functions flatten them.

Treat the Zen as a set of tie-breakers, not laws ("practicality beats purity" is in there too). When two approaches both work, it helps you explain why one is better.

PEP 8, and ruff as Its Enforcer

PEP 8 is Python's style guide. It covers naming (`snake_case` for functions and variables, `CapWords` for classes, `UPPER_CASE` for constants), whitespace, import order, line length, and a few programming recommendations such as comparing to `None` with `is`. Nobody memorizes all of it, and nobody needs to, because tools check it for you. The tool we use in this book is **ruff**, a fast linter and formatter that implements the PEP 8 checks plus hundreds of other rules taken from popular linters. Recent versions of ruff are what the `mapleridge` project is configured for.

The project's settings live in `pyproject.toml`. The relevant part looks like this:

pyproject.toml

```
[tool.ruff]
line-length = 90
target-version = "py312"
extend-exclude = ["legacy"]

[tool.ruff.lint]
select = ["E", "F", "W", "I", "B", "UP", "SIM", "RUF"]
```

The `select` line turns on several rule families: `E` and `W` are pycodestyle (PEP 8) errors and warnings, `F` is Pyflakes (unused imports, undefined names), `I` sorts imports, `B` is flake8-bugbear (likely bugs, such as the mutable default argument we meet in Chapter 2), `UP` suggests modern syntax for the target version, `SIM` suggests simplifications, and `RUF` is ruff's own rule set. The legacy folder is excluded from routine checks so the noise does not hide real problems in the package, but nothing stops us from pointing ruff at the legacy script explicitly. From inside the `sample` folder:

Terminal

```
py -3.12 -m ruff check --output-format concise legacy/legacy_report.py
```

Output

```
legacy\legacy_report.py:5:1: E401 [*] Multiple imports on one line
legacy\legacy_report.py:5:1: I001 [*] Import block is un-sorted or un-formatted
legacy\legacy_report.py:11:9: SIM115 Use a context manager for opening files
legacy\legacy_report.py:16:22: E711 Comparison to `None` should be `cond is None`
legacy\legacy_report.py:43:8: E712 Avoid equality comparisons to `True`; ...
legacy\legacy_report.py:102:64: E713 [*] Test for membership should be `not in`
legacy\legacy_report.py:145:5: E722 Do not use bare `except`
Found 7 errors.
[*] 3 fixable with the `--fix` option (2 hidden fixes can be enabled with the ...
```

(Two long lines are shortened with "..." to fit the page.) Seven findings in about 150 lines, and every one of them is an idiom this book teaches. Here is what each code means in plain words:

Code	What ruff found	Idiomatic fix	Chapter

E401, I001	`import sys, os, csv`	One import per line, sorted	26
SIM115	`open()` without `with`	`with open(...) as f:`	14
E711	`header == None`	`header is None`	1, 2
E712	`premium == True`	`if premium:`	1, 8
E713	`not t["seat"] in acc`	`t["seat"] not in acc`	3, 28
E722	bare `except:`	catch specific exceptions	1, 25

Notice what ruff does *not* find. It does not complain about `for i in range(len(screenings))`, the bubble sort, the nested loops that look up a film by scanning every row, or the strings padded with `" " * (24 - len(title))`. Linters catch mechanical problems; idioms that need understanding of the problem are your job. We run ruff anyway: it frees your attention for the parts a tool cannot see.

> **NOTE**
>
> ruff's exact messages and its default rule selection change between releases. With no configuration file, recent versions enable only a conservative default set, which is why our scratch demos pass `--select` explicitly. Inside the `sample` folder the project's `pyproject.toml` decides. The `--output-format concise` flag keeps the report to one line per finding; without it, ruff prints a small code excerpt for each problem.

A First Before and After

Let us look at some of Sam's code up close. This is how the legacy script reads a CSV file and then picks the screenings for one day:

`legacy/legacy_report.py (excerpt)`

```
def readfile(name):
    f = open(os.path.join(DATA, name), encoding="utf-8")
    rows = []
    r = csv.reader(f)
    header = None
    for line in r:
        if header == None:
            header = line
            continue
        d = {}
        for i in range(len(header)):
            d[header[i]] = line[i]
        rows.append(d)
    f.close()
    return rows

...
    todays = []
```

```
    for i in range(len(screenings)):
        if screenings[i]["starts_at"][0:10] == day:
            todays.append(screenings[i])
```

It produces the right answer, but it does a lot of work by hand that Python already does for us, and each piece of hand-made machinery is a place where a bug can hide:

- **Manual dict building.** The inner loop zips the header with each line, one index at a time. The standard library's `csv.DictReader` does exactly this, correctly, including edge cases.
- **The header flag.** `header == None` is a state machine that exists only to skip the first row. `DictReader` reads the header for us.
- **`open` without `with`.** If anything raises between `open` and `f.close()`, the file stays open. A `with` block closes it no matter what.
- **`for i in range(len(...))`.** The loop never needs `i` for anything except indexing back into the same list. Iterate over the items themselves.
- **Slicing a date string.** `starts_at[0:10] == day` depends on the exact text layout of the file. It works, but it says "compare characters" when the meaning is "starts on this day".

Here is the same job written both ways, side by side, so you can see that the results match:

demo_loops.py

```
import csv
from pathlib import Path

DATA = Path("sample/data")

# Before: the style Sam used in legacy_report.py
f = open(DATA / "screenings.csv", encoding="utf-8")
rows = []
r = csv.reader(f)
header = None
for line in r:
    if header == None:
        header = line
        continue
    d = {}
    for i in range(len(header)):
        d[header[i]] = line[i]
    rows.append(d)
f.close()
todays = []
for i in range(len(rows)):
    if rows[i]["starts_at"][0:10] == "2026-03-15":
        todays.append(rows[i])
print([s["screening_id"] for s in todays])

# After: the same result, the Pythonic way
with open(DATA / "screenings.csv", newline="", encoding="utf-8") as f:
```

```
    rows = list(csv.DictReader(f))
sunday = [row for row in rows if row["starts_at"].startswith("2026-03-15")]
print([row["screening_id"] for row in sunday])
```

Output

```
['S109', 'S110']
['S109', 'S110']
```

Eighteen lines became three. The new version opens the file with `with` (closed automatically), passes `newline=""` as the `csv` module documentation recommends, lets `DictReader` map headers to values, and uses a list comprehension (Chapter 4) to filter. `startswith` also states the intent better than a slice. The package goes one step further: `mapleridge.loaders` turns each row into a real `Screening` object with a `datetime`, so "the screenings on Sunday" becomes `cinema.schedule.on(date(2026, 3, 15))`, with no string handling at all.

Why `is None` and not `== None`

ruff flagged `header == None` as E711. This is more than taste. The `==` operator calls the left operand's `__eq__` method, and a class can define `__eq__` to return anything. The `is` operator compares identity, and there is exactly one `None` object, so `x is None` cannot be fooled:

demo_none.py

```
class AlwaysEqual:
    """A (silly) class whose __eq__ says yes to everything."""

    def __eq__(self, other):
        return True


value = AlwaysEqual()
print("value == None:", value == None)
print("value is None:", value is None)
```

Output

```
value == None: True
value is None: False
```

Real libraries do override `==` in surprising ways. Using `is None` sidesteps all of that, and it reads better. The same reasoning explains E712: `if premium == True:` should be `if premium:`, because the variable already is a truth value. Chapter 2 explains identity in depth, and Chapter 8 covers truthiness rules.

Errors should never pass silently

The legacy script ends with this:

legacy/legacy_report.py (excerpt)

```
if __name__ == "__main__":
    try:
        sys.stdout.reconfigure(encoding="utf-8")
        if len(sys.argv) > 1:
            print(main(sys.argv[1]))
        else:
            print(main("2026-03-14"))
    except:
        print("something went wrong")
```

Let us see what that does when Rosa mistypes a date:

Terminal

```
py -3.12 -X utf8 legacy/legacy_report.py 2026-13-40
```

Output

```
something went wrong
```

Something did go wrong, but what? A bad month? A missing file? A typo in a column name? A bug in the price calculation? The bare `except:` catches *everything*, including `KeyboardInterrupt` and `SystemExit`, and throws away the one piece of information that would help: the exception itself. It also exits with status 0, so a scheduled job would report success. Compare a version that catches only the error it expects and says what happened:

demo_bare_except.py

```
from datetime import date


def parse_day_legacy(text):
    try:
        return date(int(text[0:4]), int(text[5:7]), int(text[8:10]))
    except:
        print("something went wrong")


def parse_day(text: str) -> date:
    try:
        return date.fromisoformat(text)
    except ValueError as exc:
        raise SystemExit(f"error: bad --day {text!r} ({exc})") from None


print(parse_day_legacy("2026-13-40"))
print(parse_day("2026-03-14"))
print(parse_day("2026-13-40"))
```

Output

```
something went wrong
None
2026-03-14
error: bad --day '2026-13-40' (month must be in 1..12)
```

The legacy function prints its vague message and then silently returns `None`, which the caller happily prints, so the failure leaks into the next step of the program. The idiomatic version uses `date.fromisoformat` instead of slicing, catches only `ValueError`, and stops the program with a precise message and a non-zero exit status (the process exits with status 1 here). The package's command-line interface takes the same approach: it lets `argparse` parse the date with `type=date.fromisoformat`.

> **WARNING**
>
> Never write a bare `except:`, and never write `except Exception: pass`. Both turn bugs into mysteries. Catch the specific exception you expect, at the place where you know what to do about it, and let everything else propagate. Chapter 25 is devoted to doing exceptions well.

Hand-made formatting versus a real type

Sam's script stores money as integer cents, which is a good instinct (Chapter 9 explains why floats are wrong for money). But it formats those cents with a hand-written function. Let us test it against the package's `Money` type on a few values, including a refund:

demo_money.py

```
from mapleridge import Money


def money(c):  # from legacy_report.py
    s = str(c // 100) + "." + ("0" + str(c % 100))[-2:]
    if c >= 100000:
        s = s[:-6] + "," + s[-6:]
    return "$" + s


for cents in (1250, 36500, 123450, -200):
    print(f"{cents:>7}  legacy={money(cents):<10}  Money={Money(cents)}")
```

Output

```
   1250  legacy=$12.50      Money=$12.50
  36500  legacy=$365.00     Money=$365.00
 123450  legacy=$1,234.50   Money=$1,234.50
   -200  legacy=$-2.00      Money=-$2.00
```

The legacy function is fine for the amounts it was tested with and quietly wrong for a negative amount, and it would insert only one thousands separator for a million dollars. `Money.__str__` is a single f-string with the format specification `,.2f`, which handles every thousands separator, plus one line for the sign. Python's built-in formatting has already solved these problems; code that uses it inherits the solution.

Touring the mapleridge Package

Now let us look at the package you will be working on. The book's companion files contain a folder called `sample` with this layout:

sample/

```
sample/
  pyproject.toml           pytest, ruff, and mypy settings
  data/                    films.csv  screens.csv  screenings.csv  tickets.csv
                           concessions.csv  concession_sales.csv
                           ticket_requests_bad.csv
  legacy/legacy_report.py  Sam's original report script
  mapleridge/
    __init__.py   __main__.py   cli.py
    money.py      enums.py      errors.py     films.py       seating.py
    screenings.py pricing.py    tickets.py    concessions.py loaders.py
    reports.py    decorators.py resources.py  protocols.py   commands.py
  tests/
    conftest.py   test_*.py
```

Each module is small and focused. The table below says what each one holds and which chapter explains the idioms inside it. You do not need to read the code yet; this is a map for later.

Module	What it contains	Main chapters
`money.py`	`Money`, an immutable amount in integer cents	9, 10
`enums.py`	`Rating`, `TicketType`, `SeatType`, `ScreenFeature`	21
`films.py`	the `Film` dataclass	10, 20
`seating.py`	`Seat`, `SeatMap`, `Screen`, `free_blocks()`	7, 8, 12, 20
`screenings.py`	`Screening`, `Schedule`	8, 10, 11, 20
`pricing.py`	`BASE_PRICES`, `ticket_price()`	3, 15, 19
`tickets.py`	`Ticket`, `BoxOffice`	3, 13, 23, 25
`loaders.py`	CSV readers, `load_cinema()`	5, 12, 25
`reports.py`	`daily_summary()` and helpers	13, 27, 28
`decorators.py`	`log_calls`, `timed`, `retry`	17, 18
`commands.py`	a small command language using `match`	22

The rest (`errors.py`, `concessions.py`, `resources.py`, `protocols.py`, `cli.py`) appear in Chapters 25, 23, 14, 24, and 26 respectively. The package's `__init__.py` re-exports the most useful names, so in examples we write imports such as `from mapleridge import Money, load_cinema`.

Running the cinema from the command line

The package has a `__main__.py`, which means Python can run it with the `-m` switch (Chapter 26 explains the mechanics). Change into the `sample` folder and ask for help first:

Terminal

```
py -3.12 -X utf8 -m mapleridge --help
```

Output

```
usage: mapleridge [-h] {summary,films,seats} ...

Mapleridge Cinema tools

positional arguments:
  {summary,films,seats}
    summary             daily sales summary
    films               list films and ticket revenue
    seats               show free seats for a screening

options:
  -h, --help            show this help message and exit
```

Now print the same daily summary that Sam's script prints. With no `--day`, the command defaults to Saturday, March 14, 2026, the first day in the sample data:

Terminal

```
py -3.12 -X utf8 -m mapleridge summary
```

Output

```
Mapleridge Cinema - Saturday, March 14, 2026
    Time  ID    Film                     Screen  Sold    Occ.   Tickets
11:00 AM  S101  Pip and the Paper Kites   Maple      7   14.0%    $59.00
 1:30 PM  S102  The Quiet Orchard         Birch      3    9.4%    $22.50
 2:00 PM  S103  The Long Signal           Maple      3    6.0%    $32.00
 4:15 PM  S104  Café Mirabelle            Cedar      3   16.7%    $25.50
 6:00 PM  S105  Second Helpings           Birch      5   15.6%    $59.00
 7:30 PM  S106  The Long Signal           Maple      8   16.0%   $106.50
 8:00 PM  S108  Café Mirabelle            Cedar      2   11.1%    $20.00
 9:00 PM  S107  Midnight at the Paragon   Birch      3    9.4%    $40.50
Tickets sold: 34
Ticket revenue: $365.00
Concession revenue: $195.25
Total revenue: $560.25
```