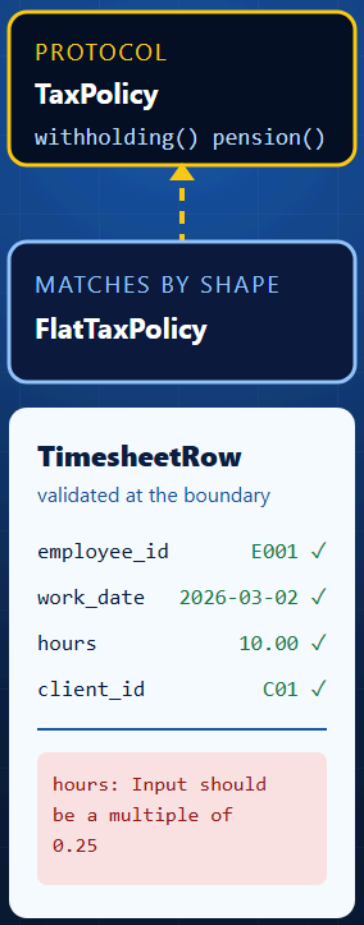
wrenhall/payroll.py

```
def compute_payslip(
    employee: Employee,
    entries: Iterable[TimesheetEntry],
    period_start: date,
    policy: TaxPolicy = DEFAULT_POLICY,
) -> Payslip:
```

```
> py -3.12 -m mypy --strict wrenhall
pay.py:14: error: Unsupported operand
types for * ("None" and "int") [operator]
Success: no issues found in 13 source files
```

```
type Money = Decimal
to_money("104.475") # 104.48
```



A COMPLETE, PRACTICAL GUIDE

Typed Python

Type Hints, mypy, Protocols, Generics, and Data Validation with Pydantic

Annotations · Optional · TypedDict · Dataclasses · NewType
Protocols · PEP 695 Generics · ParamSpec · Stubs · Pydantic v2

STATIC TYPES · RUNTIME VALIDATION · PYTHON 3.12

THENEXBOOKS

Typed Python

Type Hints, mypy, Protocols, Generics, and Data Validation with Pydantic
From your first annotation to a strictly typed, validated codebase — built step by step

First Edition

THENEXBOOKS

Typed Python
Type Hints, mypy, Protocols, Generics, and Data Validation with Pydantic

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, mypy, Pydantic, Ruff, and the other tools in this book change over time; command options, error messages, and defaults can differ between versions. Always consult the documentation for the versions you use.

Python is a registered trademark of the Python Software Foundation. Pydantic, Ruff, PyCharm, Visual Studio Code, GitHub, and other product names are trademarks 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 or any of the projects and companies mentioned.

Wrenhall Payroll, the payroll and invoicing service used as the running example in this book, is fictional, as are its employees, clients, timesheets, and all sample data. Its pay, overtime, tax, and pension rules are deliberately simplified for teaching and are not real tax or employment law. Email addresses use reserved example domains.

Preface

Python became popular partly because it does not make you declare types. You can write a function in a few lines, run it, and see what happens. That freedom is wonderful for a script and dangerous for a system. As a codebase grows, the questions multiply: can this argument be `None`? Is this amount a `float`, a `Decimal`, or a string that came straight from a CSV file? Which of these three dictionaries has a `"rate"` key? Every one of those questions is answered either by reading a lot of code or by finding out in production.

Type hints let you write the answers down, in the code, where tools can check them. A static type checker such as mypy reads those hints and reports mistakes before the program runs. A validation library such as Pydantic uses the same hints to check data that arrives at runtime, from files, environment variables, and other systems. Together they turn a loose collection of functions into a codebase whose shape you can trust.

This book teaches typed Python from the first annotation to a strictly checked, validated service. We do not just list the contents of the `typing` module. We build one real program step by step, introduce each feature at the moment it solves a problem we actually have, run the checker, read its error messages, and fix them.

Who This Book Is For

- **Python developers** who write working Python every day and want to start using type hints properly instead of decoratively.
- **Teams adopting mypy** on an existing codebase who need a realistic, gradual path from "no hints" to strict mode.
- **Pydantic users** who want to understand what their models actually do: validation modes, validators, serialization, and JSON Schema.
- **Reviewers and tech leads** who need shared conventions for `Any`, `type: ignore`, stubs, and CI checks.

We assume you know Python itself: functions, classes, modules, exceptions, virtual environments, and pip. We do not teach programming basics or object-oriented design.

What You Will Learn

- Why type hints matter, what they do and do not do, and how to set up Python 3.12, mypy, and Ruff for a project.
- How to annotate functions and variables, handle `None` safely, and type collections, tuples, and `TypedDict`.
- How to model a domain with `Literal`, enums, `Final`, typed dataclasses, `NewType`, type aliases, and money as `Decimal`.

- How to type classes and inheritance, use `Self` and `@override`, and narrow types with `isinstance`, `match`, `TypeGuard`, and `TypeIs`.
- How to write structural interfaces with Protocols, reusable code with generics and the new Python 3.12 syntax, and fully typed decorators with `ParamSpec`.
- How to use overloads and the escape hatches (`Any`, `object`, `cast`, `type: ignore`) without losing safety.
- How to configure mypy, type third-party code with stubs, adopt types gradually in a legacy codebase, and read any mypy error with confidence.
- How to validate untrusted input with Pydantic v2: models, constraints, validators, serialization, JSON Schema, settings, CSV, JSON, and config files.
- How types behave at runtime, what they cost, and how to keep a team's codebase typed with conventions, tests, and CI.

How This Book Is Organized

Part I — Foundations of Typed Python (Chapters 1–5) explains why types help, sets up the tools, and covers annotations, `None` and unions, and collections. **Part II — Modeling the Domain with Types** (Chapters 6–10) covers literals, enums, dataclasses, money as `Decimal`, domain types, typed classes, and type narrowing. **Part III — Advanced Static Typing** (Chapters 11–15) covers Protocols, generics, the PEP 695 syntax of Python 3.12, callables and decorators, overloads, and escape hatches. **Part IV — mypy in Practice** (Chapters 16–20) covers configuration, stubs, gradual typing of legacy code, iterators and async code, and a field guide to mypy errors. **Part V — Data Validation with Pydantic** (Chapters 21–24) covers models, validators, serialization, JSON Schema, settings, and validating real input. **Part VI — Types in Production** (Chapters 25–28) covers types at runtime, performance, team conventions and CI, and a complete case study. The appendices provide a typing cheat sheet, templates and reference tables, a production checklist, and a glossary.

Each chapter ends with exercises and a **Key Takeaways** box for quick review.

The Running Example: Wrenhall Payroll

Throughout the book we type and validate **Wrenhall Payroll**, the internal payroll and invoicing service of Wrenhall, a fictional small software consultancy. Four people work there — Ada Brennan, Tomas Reyes, Priya Nair, and Marcus Hale — and they log their hours each week against two clients, Harrowgate Dental and Lumen Brewing Co. Every Monday the service runs payroll for the previous week, with overtime for hourly staff, withholding and pension deductions, and payslips, and it invoices each client for billable hours plus sales tax. Money is always a `Decimal`, rounded to the cent. The data is small enough to check by hand, and every number in the book comes from running the code.

We start with a few loosely typed functions and finish with a package called `wrenhall` that passes `mypy --strict`, validates every file and payload it reads with Pydantic, and has tests and CI checks that keep it that way.

Versions and Conventions

Examples target **Python 3.12**, checked with a recent version of **mypy**, validated with **Pydantic v2**, and linted with **Ruff**. Features introduced in Python 3.13 and 3.14 are labeled as such, and we show how to use them on 3.12 through the `typing_extensions` package where that is possible. Another popular type checker, pyright, is discussed where its behavior differs, but all checker output in this book comes from mypy. Commands are shown for Windows (`py -3.12 -m mypy`); on macOS and Linux use `python3 -m mypy` or the command from your virtual environment.

> **TIP**
>
> A recommended practice or a technique that makes your work easier.

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

> **WARNING**
>
> A common mistake or something that can cause bugs, data errors, or security problems.

Code appears in `monospace`. A caption above each listing names the file it belongs to, such as `wrenhall/payroll.py`; **Terminal** marks a command and what it printed, and **Output** marks what a program printed. All program output and all mypy messages were produced by actually running the code; a few very long lines are wrapped to fit the page.

Contents

Preface

PART I — FOUNDATIONS OF TYPED PYTHON

1 Why Type Hints? Static Types in a Dynamic Language

2 Setting Up: Python 3.12, mypy, Ruff, and Your Editor

3 Annotating Variables and Functions

4 None, Optional, and Union: Handling Absence Safely

5 Collections, Tuples, and TypedDict

PART II — MODELING THE DOMAIN WITH TYPES

6 Literal, Enum, and Final: Constraining Values

7 Typed Dataclasses: Records, Value Objects, and Money as Decimal

8 NewType, Type Aliases, and Domain Types

9 Typing Classes: Inheritance, ClassVar, Self, and @override

10 Type Narrowing: isinstance, match, TypeGuard, and TypeIs

PART III — ADVANCED STATIC TYPING

11 Protocols: Structural Typing and Duck Typing Done Right

12 Generics and TypeVar: Writing Reusable Typed Code

13 Python 3.12 Generic Syntax: PEP 695 and the type Statement

14 Callables, ParamSpec, and Typed Decorators

15 Overloads, Any, object, Never, and cast: Precision and Escape Hatches

PART IV — MYPY IN PRACTICE

16 Configuring mypy: Strict Mode, Per-Module Settings, and Error Codes

17 Typing Third-Party Code: Stubs, py.typed, and typeshed

18 Gradual Typing a Legacy Codebase

19 Typing Iterators, Generators, Context Managers, and Async Code

20 Reading and Fixing mypy Errors: A Field Guide

PART V — DATA VALIDATION WITH PYDANTIC

21 Pydantic v2 Models: Types That Validate at Runtime

22 Validators, Constraints, and Custom Types

23 Serialization, JSON Schema, and Settings

24 Validating Real Input: CSV Files, JSON APIs, and Config Files

PART VI — TYPES IN PRODUCTION

25 Types at Runtime: Annotations, Introspection, and Runtime Checks

26 Performance: What Types and Validation Really Cost

27 Team Conventions and CI: mypy, Ruff, Tests, and Type Coverage

28 Case Study: The Fully Typed Wrenhall Payroll Service

APPENDICES

A Typing Cheat Sheet

B Templates and Reference

C Production Checklist

D Glossary

PART I

Foundations of Typed Python

What type hints add to a dynamic language and what they do not, setting up Python 3.12, mypy, and Ruff, annotating functions and variables, handling None safely, and typing the collections every program uses.

1 · Why Type Hints?
2 · Setting Up
3 · Annotating Variables and Functions
4 · None, Optional, and Union
5 · Collections, Tuples, and TypedDict

CHAPTER 1 · PART I — FOUNDATIONS OF TYPED PYTHON

Why Type Hints? Static Types in a Dynamic Language

Python never asks you to declare a type, so why would you want to? In this chapter we watch a small payroll script go wrong in two quiet ways, fix it with a handful of annotations and the `Decimal` type, run a type checker for the first time, and meet Wrenhall Payroll, the company whose code we will type from the first page of this book to the last.

A Language That Trusts You

Python is a dynamically typed language. A variable is just a name that points at an object, and the object carries its own type. A function happily accepts whatever you pass it, and nothing is checked until an operation actually runs. If the operation makes sense for the objects involved, it succeeds; if not, Python raises a `TypeError` or an `AttributeError`, and only then.

This flexibility is a big part of why Python is pleasant to write. For scripts, experiments, and small programs owned by one person, that freedom costs very little.

The cost shows up later, as programs grow. A function written two years ago by a colleague who has since moved on takes a parameter called `employee`. Is that an `Employee` object, a dictionary loaded from JSON, or an employee ID string? The only ways to find out are to read every caller, read the body, or run the code and see what breaks.

Type hints are Python's answer to that problem. They let you *say*, in the source code, what kinds of values a function expects and returns, what a variable holds, and what shape a record has. A separate tool, the type checker, then reads your code without running it and tells you where those statements are contradicted. This book is about writing those statements well, reading what the checker tells you, and, in the second half, using the same annotations to validate data at runtime with Pydantic.

The First Bug: A String Where a Number Belongs

Let us start with code rather than theory. Wrenhall, the small software consultancy we introduce properly later in this chapter, once paid its staff with a script much like the following. It reads hours from a CSV file and multiplies them by an hourly rate. Here is the input file, with one employee who worked 46.5 hours:

hours.csv

```
name,hours
```

```
Ada,46.5
```

And here is the script. It has no annotations at all, which is how most Python code in the world still looks:

pay_v0.py

```
import csv


def calc_gross(hours, rate):
    return hours * rate


with open("hours.csv", newline="") as f:
    for row in csv.DictReader(f):
        print(row["name"], calc_gross(row["hours"], 42))
```

Run it and look closely at the result. We trimmed the line, because it goes on for a long time:

Terminal

```
> py -3.12 pay_v0.py
Ada 46.546.546.546.546.546.546.546.546.546.546.546.546.546.546.546.546.546.546.546.546.5...
```

There is no exception and no warning. The program "worked." The trouble is that `csv.DictReader` returns every field as a string, so `row["hours"]` is the text `"46.5"`, not a number. In Python, multiplying a string by an integer repeats the string, so `"46.5" * 42` is a perfectly legal expression that produces forty-two copies of the text. Ada's gross pay is now a 168-character string. If this value had gone into a database column or a payment file instead of a terminal, someone might not have noticed until a bank rejected it, or worse, until it was silently parsed as something else.

This is the most important kind of bug that type hints help with: not the crash, which at least announces itself, but the wrong-but-legal operation that slips through because Python's operators are generous. A crash tells you exactly where it hurts. A wrong value travels.

The obvious fix, and the second bug

The quick fix most of us would type is to convert the field with `float()`. The script now prints a number, and it even looks right:

pay_v0b.py (excerpt)

```
        print(row["name"], calc_gross(float(row["hours"]), 42.0))
```

Output

```
Ada 1953.0
```

For these two numbers the binary floating-point answer happens to be exact. That luck does not last. The `float` type stores numbers in base 2, and most decimal fractions, including the humble 0.1, have no exact base-2 representation. A few lines in the REPL show what that means for money:

Python REPL

```
>>> 0.1 + 0.2
0.30000000000000004
>>> 0.1 + 0.2 == 0.3
False
>>> 1.10 * 3
3.3000000000000003
>>> round(2.675, 2)
2.67
```

The last line surprises almost everyone. Mathematically 2.675 lies exactly halfway between 2.67 and 2.68, and ordinary commercial rounding would give 2.68. But the float literal `2.675` is actually stored as a value very slightly *below* 2.675, so it rounds down. In a payroll system these errors are a cent here and a cent there, and they accumulate across employees, weeks, and invoices until a total no longer matches the sum of its lines. Accountants notice that kind of thing.

Python's standard library has a type built for exactly this job: `decimal.Decimal`. A `Decimal` created from a string stores the decimal digits you wrote, arithmetic on it follows decimal rules, and you choose the rounding mode explicitly:

Python REPL

```
>>> from decimal import Decimal, ROUND_HALF_UP
>>> Decimal("0.1") + Decimal("0.2") == Decimal("0.3")
True
>>> Decimal("2.675").quantize(Decimal("0.01"), rounding=ROUND_HALF_UP)
Decimal('2.68')
>>> Decimal(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
```

Notice the last line: building a `Decimal` from a float faithfully copies the float's error. The rule for money is therefore twofold. Use `Decimal`, and build it from strings or integers, never from floats. Chapter 7 goes deeper into `Decimal`, quantizing, and rounding modes. For now the point is simpler: "use Decimal, not float" is a rule that lives in people's heads. A rule that lives in people's heads is broken the first time someone is in a hurry. We want a tool that enforces it.

> **WARNING**
>
> Never represent money as `float`, and never build a `Decimal` from a float literal such as `Decimal(0.1)`. Both look harmless in a small test and both produce off-by-a-cent errors in real totals. Throughout this book money is always `Decimal` built from strings or integers.

Saying What You Mean with Annotations

A type hint, or annotation, is a piece of syntax that attaches a type to a parameter, a return value, or a variable. Function annotations have been part of the language since Python 3.0, and the `typing` module and the meaning we use today were standardized in PEP 484 for Python 3.5. Here is the same function with annotations:

pay.py

```
import csv
from decimal import Decimal


def calc_gross(hours: Decimal, rate: Decimal) -> Decimal:
    return hours * rate


with open("hours.csv", newline="") as f:
    for row in csv.DictReader(f):
        print(row["name"], calc_gross(row["hours"], 42))
```

The colon after a parameter name introduces its type; the arrow `->` before the final colon gives the return type. Read aloud, the signature says: "`calc_gross` takes two decimals and returns a decimal." That sentence is now part of the code, where every reader and every tool can see it, instead of living in a docstring that may or may not be up to date.

Run the program as it is, still passing a string and an integer, and nothing changes. You get the same forty-two copies of `46.5`. That is not an accident and it is not a bug in Python. It is the single most important fact about type hints, so it gets its own section.

What Annotations Do Not Do

CPython, the standard Python interpreter, does not enforce annotations. When it executes a `def` statement, it evaluates the annotations and stores them in a dictionary on the function object, and that is all. When the function is called, no one compares the arguments with the stored types. You can see both halves of that behavior in a few lines:

Python REPL

```
>>> def add(a: int, b: int) -> int:
...     return a + b
...
>>> add("4", "2")
'42'
>>> add.__annotations__
{'a': <class 'int'>, 'b': <class 'int'>, 'return': <class 'int'>}
```

The function claimed to add integers and cheerfully concatenated two strings. The annotations are right there in `__annotations__`, available to any library that wants to read them, but the interpreter itself ignores them when calling the function. This leads to three facts worth stating plainly, because each one is a common misunderstanding:

- **Annotations are not runtime checks.** A wrong argument is not rejected at the call. If you want runtime checking, a library has to do it for you; Pydantic, which we use in Part V, is one such library.
- **Annotations do not make CPython faster.** The interpreter does not use them to optimize anything. The only runtime cost is evaluating and storing them when the function or class is defined, which is tiny. Chapter 26 looks at real costs, including the cost of runtime validation.
- **Annotations are optional and gradual.** You can annotate one function in a large codebase and leave the rest alone. Unannotated code keeps working exactly as before. Chapter 18 turns this into a strategy for adopting types in an existing project.

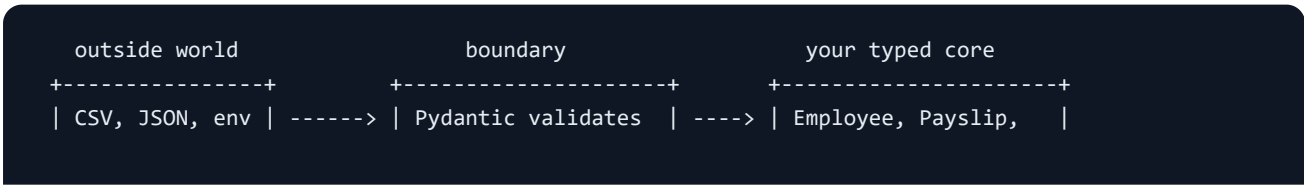
So who gives annotations their meaning? Two kinds of tools, and they correspond to the two halves of this book.

Two Kinds of Checking: Static and Runtime

A **static type checker** reads your source code without running it. It builds a model of every name and expression in the program, works out what type each one has (using your annotations where you wrote them and its own inference where you did not), and reports every place where the code contradicts itself: a string passed where a `Decimal` is expected, an attribute that does not exist, a value that might be `None` used as if it never could be. The reference checker is **mypy**, the tool this book uses throughout. Other checkers exist, notably pyright, which powers Pylance in VS Code; Chapter 2 compares them briefly.

A **runtime validator** runs while your program runs. It looks at actual values, typically data coming from outside the program (a CSV file, a JSON document, environment variables), and checks them against a declared shape, converting and rejecting as needed. The library we use for this is **Pydantic** (version 2), which reads the same annotation syntax you write for mypy and turns it into validation logic.

The two are complementary, not competing. The static checker proves that your code is consistent with itself: if `split_hours` receives a `Decimal`, every caller must pass one. But it cannot know what is in a file that does not exist yet. The runtime validator guards the border where untrusted data enters the program, so that once data is inside, it really does have the types the checker assumed. The diagram below is the mental model we use for the rest of the book.



```
  | (untrusted)    |  text   | and converts        | typed | Decimal money ...    |
  +----------------+         +---------------------+       +----------------------+
                                    runtime check            checked statically by mypy
```

Part I to Part IV of the book are about the right side of that picture: the static type system and mypy. Part V is about the middle box. Part VI ties them together in production.

Running a Type Checker for the First Time

Let us give our annotated `pay.py` to mypy. Chapter 2 covers installation and configuration in detail; for now it is enough to know that on our machine mypy is installed for Python 3.12 and runs as a module. Open a terminal in the folder that contains `pay.py` and type:

Terminal

```
> py -3.12 -m mypy pay.py
pay.py:11: error: Argument 1 to "calc_gross" has incompatible type "str | Any";
    expected "Decimal"  [arg-type]
pay.py:11: error: Argument 2 to "calc_gross" has incompatible type "int"; expected
    "Decimal"  [arg-type]
Found 2 errors in 1 file (checked 1 source file)
```

(Throughout the book, when a real message is longer than the page allows, we wrap it at a space and indent the continuation by four spaces. Nothing else in the output is changed.)

Mypy found both problems without running the program, without the CSV file even needing to be correct, and without a single test. Let us read the first message the way you will read hundreds of them in this book:

- `pay.py:11` is the file and line number.
- `error` is the severity. Mypy also prints `note` lines that add context to an error.
- `Argument 1 to "calc_gross" has incompatible type "str | Any"; expected "Decimal"` is the message. Mypy knows the values of a `DictReader` row are strings (or, in some cases the library's stubs cannot rule out, anything at all, which is what `Any` means), and it knows the parameter wants a `Decimal`. The two do not match.
- `[arg-type]` is the **error code**. Every mypy error has one. Codes let you search the documentation, configure mypy per code, and silence one specific error on one line when you truly must. Chapter 20 is a field guide organized by these codes.

The second message is just as interesting. We passed the integer `42` where a `Decimal` was expected. Python would multiply a `Decimal` by an `int` without complaint, so why does mypy object? Because we *said* the rate is a `Decimal`, and an `int` is not a `Decimal`. The checker holds us to our own declaration, which is exactly what we want: the rule "rates are decimals" is now enforced everywhere the function is called.

Now fix the call site by converting at the boundary, where the text from the file enters the program:

pay.py (fixed loop)

```
with open("hours.csv", newline="") as f:
    for row in csv.DictReader(f):
        print(row["name"], calc_gross(Decimal(row["hours"]), Decimal("42.00")))
```

Terminal

```
> py -3.12 -m mypy pay.py
Success: no issues found in 1 source file
> py -3.12 pay.py
Ada 1953.000
```

That is the whole loop we follow in every chapter: write code, run mypy, read the error, fix it, run mypy again, then run the program. The output, `1953.000`, is a `Decimal` that kept the three decimal places produced by multiplying one-decimal hours by two-decimal rates. Rounding money to cents is a separate decision we make explicitly with a helper called `to_money()`, which you will meet in a moment.

> **NOTE**
>
> We show commands as typed on Windows with the `py` launcher: `py -3.12 -m mypy pay.py`. On macOS and Linux the equivalent is `python3 -m mypy pay.py`, or just `python -m mypy pay.py` inside an activated virtual environment. Running tools with `-m` guarantees they use the same interpreter as your code.

What Static Typing Gives You

Catching the string-times-integer bug early is the most visible benefit, but not the only one.

Documentation that cannot go stale

A docstring that says "hours: number of hours as a Decimal" is a promise nobody checks. An annotation `hours: Decimal` is checked on every run of mypy. When someone changes the function to accept `int` too, they must change the annotation, or mypy complains at every call site that now disagrees. Annotations are the one form of documentation that is verified by a machine.

Better editor support

Editors use the same type information for accurate completions, signatures, and instant error highlighting. Typing `payslip.` lists the real fields of a `Payslip` rather than a guess.

Refactoring with a safety net

Rename a field, change a return type from a tuple to a dataclass, split one function into two. In an untyped codebase you search with grep and hope. In a typed codebase you make the change, run mypy, and get a precise list of every place that needs updating. This is often the moment a team

becomes convinced: a refactoring that would have taken a nervous afternoon takes twenty minutes and a clean mypy run.

Bugs found before the code runs

Mypy finds mistakes in code paths your tests never exercise: the error branch, the rarely used option, the case where a value is `None`. It does not replace tests, because it knows nothing about whether your overtime rule is the right business rule, but it covers a different class of mistakes than tests do, and it covers all of the code all of the time.

What Static Typing Does Not Give You

It is just as important to be honest about the limits, so you know when to reach for something else.

- **It does not check data from outside.** Mypy trusted that `Decimal(row["hours"])` produces a `Decimal`, which it does, but if the file contains `"forty"` the conversion raises at runtime. Validating input is the job of Part V.
- **It does not check business logic.** A function that multiplies hours by the wrong rate is perfectly well typed. Tests, reviews, and clear code remain essential.
- **It can be switched off, a little at a time.** The type `Any` and comments such as `# type: ignore` tell the checker to stop looking. They are useful escape hatches, and Chapter 15 teaches you to use them sparingly, because every one is a small hole in the net.
- **It is only as good as the annotations it sees.** Unannotated functions are, by default, treated leniently. Chapter 3 shows exactly what mypy does and does not check inside them, and Chapter 16 shows how strict mode closes that gap.

> **TIP**
>
> A useful way to think about the trade-off: tests check that your code does the right thing for the examples you wrote; a type checker checks that your code is internally consistent for *every* value of the declared types. Neither replaces the other, and together they catch far more than either alone.

A Second Look: the Checker Reads the Whole Program

The `calc_gross` example fit on one screen. Real bugs usually cross module boundaries: a value is created in one file, passed through two functions, and misused in a third. Mypy follows the types across all of them. Here is a small probe written against the Wrenhall package (you will build the package over the next chapters; the finished version is in the book's `sample` folder). It makes two mistakes that look reasonable when you write them:

probe.py

```
from decimal import Decimal

from wrenhall.data import SAMPLE_EMPLOYEES
from wrenhall.payroll import split_hours

ada = SAMPLE_EMPLOYEES[0]
regular, overtime = split_hours(46.5)
pay = overtime * ada.hourly_rate * Decimal("1.5")
print(regular, overtime, pay)
```

Terminal

```
> py -3.12 -m mypy --strict probe.py
probe.py:7: error: Argument 1 to "split_hours" has incompatible type "float"; expected
    "Decimal"  [arg-type]
probe.py:8: error: Unsupported operand types for * ("Decimal" and "None")  [operator]
probe.py:8: note: Right operand is of type "Decimal | None"
Found 2 errors in 1 file (checked 1 source file)
```

The first error is our float bug again, caught at a call into another module. The second is subtler. `Employee.hourly_rate` is declared as `Decimal | None`, because salaried employees have no hourly rate. Mypy refuses to let us multiply by it until we have handled the `None` case. The `note` line explains why: the right operand might be `None`. Chapter 4 is entirely about this pattern, which is one of the most common sources of real crashes in Python programs.

For comparison, here is what happens if we skip the checker and just run the probe:

Terminal

```
> py -3.12 probe.py
Traceback (most recent call last):
  File "C:\work\wrenhall\probe.py", line 7, in <module>
    regular, overtime = split_hours(46.5)
                        ^^^^^^^^^^^^^^^^
  File "C:\work\wrenhall\wrenhall\payroll.py", line 25, in split_hours
    return threshold, total - threshold
                      ~~~~~~^~~~~~~~~~~
TypeError: unsupported operand type(s) for -: 'float' and 'decimal.Decimal'
```

Python stops at the first problem, deep inside `payroll.py`, and never reaches line 8. Mypy reported both problems at once, each at the line where the mistake was made rather than where it finally exploded.

Finally, the whole reference package passes the strictest standard setting mypy offers:

Terminal

```
> py -3.12 -m mypy --strict wrenhall
Success: no issues found in 13 source files
```

Meet Wrenhall Payroll

Every example in this book comes from one running project, so that each new idea lands in code you already recognize. **Wrenhall** is a small, fictional software consultancy. It has four staff, two clients, and an internal payroll-and-invoicing service called **Wrenhall Payroll**. Each week the service does two jobs: it runs payroll for the staff, producing a payslip for each person, and it invoices the clients for billable hours. All amounts are in US dollars. The code lives in a Python package named `wrenhall`.

The people

ID	Name	Pay type	Pay rate	Bill rate
E001	Ada Brennan	hourly	$42.00/h	$95.00/h
E002	Tomas Reyes	hourly	$38.50/h	$85.00/h
E003	Priya Nair	salaried	$98,800.00/yr	$120.00/h
E004	Marcus Hale	hourly	$27.25/h	none (not billable)

Three of the four are paid by the hour; Priya is salaried. Three are billable to clients at the bill rate shown; Marcus does internal office work and is never billed. Notice how much type information hides in that small table already: some people have an hourly rate and no salary, one has a salary and no hourly rate, one has no bill rate at all. In code, those "no value" cases become `None`, and handling them safely is the subject of Chapter 4.

The clients

Wrenhall has two clients: **C01 Harrowgate Dental** (accounts@harrowgate.example) and **C02 Lumen Brewing Co.** (billing@lumenbrewing.example). Employee IDs look like `E001`, client IDs like `C01`. Both are plain strings at runtime, yet mixing them up is a classic bug, and Chapter 8 makes them distinct types so that mypy rejects an employee ID where a client ID belongs.

The week

Pay periods run Monday to Sunday. Our sample week is **2026-03-02 (Monday) to 2026-03-08 (Sunday)**, and the invoices for that week are issued on Monday 2026-03-09, due 30 days later on 2026-04-08. Staff logged 18 timesheet entries that week, 169.75 hours in total. Some entries name a client and are billable; others, such as team training or office administration, are internal.

The rules

The business rules are deliberately simple:

- Hourly staff earn their hourly rate for the first 40 hours in a week and 1.5 times that rate for every hour above 40.

- Salaried staff earn one fifty-second of their annual salary every week, regardless of hours, with no overtime.
- From each gross amount, 20% is withheld and a further 5% goes to a pension; the rest is net pay.
- Invoices have one line per billable employee per client, priced at the employee's bill rate, plus 8.25% sales tax, due 30 days after issue.
- Every computed money amount is rounded to whole cents, with halves rounded up.

> **NOTE**
> Wrenhall's rates and rules are fictional and simplified on purpose so that the code stays short and the numbers are easy to check by hand. They are not real tax law for any country, state, or city, and nothing in this book is payroll or tax advice. Real payroll needs a professional and a real tax engine.

The output we are building toward

The finished package can be run as a program. For the sample week it prints the pay run and both invoices:

Terminal

```
> py -3.12 -m wrenhall
Pay run PR-2026-03-02 (2026-03-02 to 2026-03-08)
ID   Name           Reg h  OT h      Gross  Withheld  Pension        Net
E001 Ada Brennan    40.00  6.50  $2,089.50   $417.90  $104.48  $1,567.12
E002 Tomas Reyes    38.00  0.00  $1,463.00   $292.60   $73.15  $1,097.25
E003 Priya Nair     44.00  0.00  $1,900.00   $380.00   $95.00  $1,425.00
E004 Marcus Hale    40.00  1.25  $1,141.09   $228.22   $57.05    $855.82
Total gross $6,593.59, total net $4,945.19

Invoice INV-2026-0001 for Harrowgate Dental (due 2026-04-08)
  Ada Brennan   30.00 h x   $95.00 =  $2,850.00
  Tomas Reyes   20.00 h x   $85.00 =  $1,700.00
  Priya Nair    24.00 h x  $120.00 =  $2,880.00
  Subtotal $7,430.00  Tax $612.98  Total $8,042.98

Invoice INV-2026-0002 for Lumen Brewing Co. (due 2026-04-08)
  Ada Brennan   12.50 h x   $95.00 =  $1,187.50
  Tomas Reyes   18.00 h x   $85.00 =  $1,530.00
  Priya Nair    10.00 h x  $120.00 =  $1,200.00
  Subtotal $3,917.50  Tax $323.19  Total $4,240.69
```

Our worked example: Ada's overtime week

One employee gets special attention throughout the book: **Ada Brennan (E001)**, because her week exercises the most rules at once. She logged 46.50 hours: 30.00 for Harrowgate Dental, 12.50 for Lumen Brewing, and 4.00 of internal team training. Her pay works out like this:

- Regular pay: 40.00 hours x $42.00 = $1,680.00.