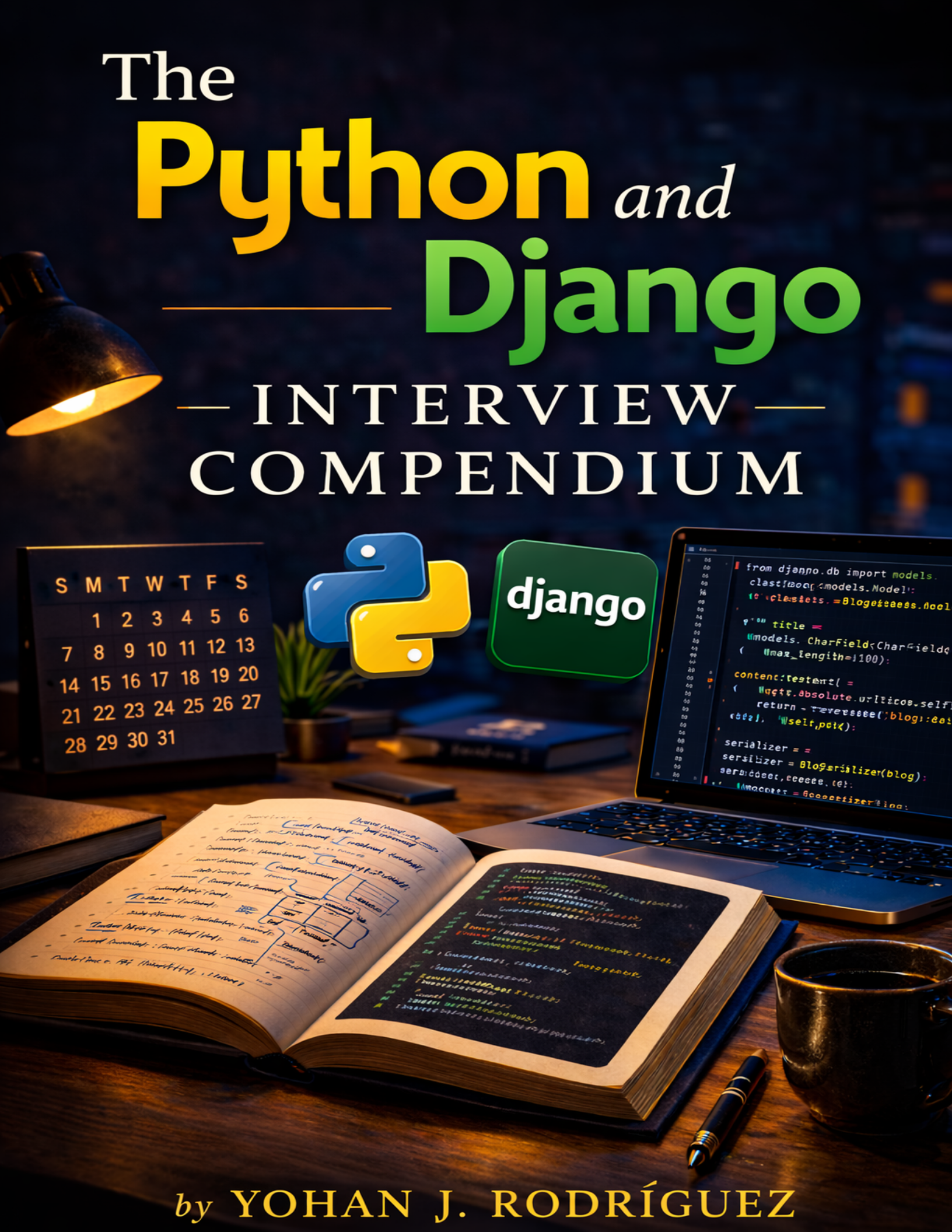
The Python and Django

INTERVIEW COMPENDIUM



by YOHAN J. RODRÍGUEZ

Preface

"Simple is better than complex."
Tim Peters

Python and Django interviews rarely stop at syntax. A single conversation can run from why a mutable default argument remembers its value between calls, to what the Global Interpreter Lock does and does not prevent, to why looping over a Django QuerySet quietly issues a hundred queries, to what you would change before a Django service takes real traffic. The language looks friendly on the surface, and the framework hides a great deal, which is exactly why interviewers probe what sits underneath. This book exists to prepare you for that conversation. It is a question-and-answer reference of more than two hundred questions across fifteen chapters, and each answer goes past the definition to the mechanism behind it, the trade-off it carries, and the way a strong candidate would explain it out loud. Almost every answer is paired with a code listing.

It is written for developers who already build things with Python, Django, or both, and now need to explain what they know. That includes mid-level engineers preparing for senior backend interviews, Python developers who use frameworks every day but have not revisited the language's data model, developers moving into Django from another stack, and Django developers who use the ORM and Django REST Framework daily and want to explain how they behave under load. It is not a first tutorial. There is no installation walkthrough and no application built step by step; Chapter 2 opens with the idea that everything in Python is an object, and the Django chapters begin from what a project and an app are rather than from a blank terminal. A reader new to Python will get more from an introductory text first. The book also stays within its subject: it does not teach other web frameworks or data-science libraries.

What you gain is concrete. You should be able to explain how Python actually behaves: identity and equality, mutability and hashing, scope and closures, iterators and generators, dunder methods, method resolution order, dataclasses and protocols, and what the GIL means when you choose between threads, processes, and `asyncio`. You should be able to read and write typed Python and test it with pytest. On the Django side, you should be able to trace a request through middleware, URL routing, and views; reason about lazy QuerySets, the N+1 problem,

`select_related` versus `prefetch_related`, and migrations that are safe to run on a live database; design Django REST Framework APIs with sensible serializers, permissions, pagination, and throttling; and discuss authentication, session and CSRF protection, caching, background tasks, and scaling in terms of trade-offs.

The book moves in two halves. The first eight chapters, about 45 percent of the pages, cover Python: its history and runtime model, fundamentals and the data model, functions and decorators, collections, iterators, and generators, object-oriented Python, exceptions, context managers, and packaging, typing, testing, and code quality, and concurrency with `asyncio` and background work. The last seven chapters cover Django: foundations, project structure, and settings; models, the ORM, migrations, and query optimization; views, forms, templates, and the admin; Django REST Framework; authentication, middleware, and security; caching, performance, queues, and scalability; and finally deployment, observability, and production troubleshooting. The halves are connected on purpose. Django is Python used with discipline, so the Django chapters keep returning to decorators, mixins, generators, and context managers, and the Python chapters point ahead to where those ideas surface in the framework.

Every chapter has the same shape. Questions are grouped into sections that move from core concepts to deeper mechanics and scenario-style questions, and many answers close with a short note on a common pitfall or a senior-level follow-up. The chapter introductions describe what interviewers tend to ask at junior, mid, and senior levels, so you can see where a topic sits. In Chapters 1 through 8 the listings are, for the most part, self-contained scripts that you can run, change, and break on purpose. In Chapters 9 through 15 they are focused Django excerpts, such as models, views, settings, serializers, and configuration, and they assume a Django project around them. Where a topic matters in more than one setting, such as N+1 queries, rate limiting, or Celery tasks, it returns in each chapter from that chapter's angle.

You can read straight through or by need. A sequential first pass builds the vocabulary that the Django half leans on, including the MRO, decorators, generators, and context managers, but each chapter is self-contained enough to be consulted on its own, and the section headings make the book easy to scan before an interview. Whichever way you read, use it actively: read a question, answer it aloud before reading further, then compare, and run the listing to see the behaviour for yourself.

A few assumptions are worth stating. You need working knowledge of Python syntax and basic programming. The Django half reads more easily if you have built a small Django application, and comfort with HTTP and basic SQL helps throughout; several answers, including full-text search and connection pooling, assume PostgreSQL. The book follows Python from 3.8 through 3.13 and calls out the release in which a feature arrived, and it does the same for Django features,

so you can answer for the version your team actually runs.

By the end, you should be able to explain, compare, and defend Python and Django decisions in an interview or a code review: what a construct does, why it behaves that way, what it costs, and when you would choose something else. No book can promise an offer, and none replaces building things. What this one can do is make the reasoning behind each answer your own, so that the next question, including one you have not seen before, has somewhere to start.

Yohan J. Rodríguez
www.imabit.com

Contents

Preface i

1 Python Fundamentals and Data Model **1**

1.1 Objects, Identity, and Types 1

1.2 Mutability, Collections, and Truthiness 4

1.3 Scope, Strings, and Copies 8

1.4 Sets, Numbers, and Unpacking 16

1 | Python Fundamentals and Data Model

Python's power comes in large part from a coherent and elegant data model. Every value in Python is an object with an identity, a type, and a value. Understanding this model is not just academic — it explains why certain operations behave the way they do, why bugs appear in code that looks correct, and why Python handles equality, copying, and mutability in ways that surprise developers coming from other languages.

Fundamentals questions are common in every round of Python interviews, from junior to senior levels. The difference is depth: a junior candidate might know that a list is mutable and a tuple is not. A senior candidate can explain why that matters for hashing, thread safety, and performance, and can describe exactly what happens in memory when you append to a list.

This chapter covers the core concepts that form the foundation of all Python programs: objects and identity, mutability and immutability, the distinction between common built-in types, and how Python evaluates truthiness. Mastering these topics will make the rest of the language significantly more intuitive.

1.1 Objects, Identity, and Types

What does it mean that everything in Python is an object?

In Python, every value — integers, strings, functions, classes, modules — is an object. An object is an instance of some class, and it carries three pieces of information: its identity (unique memory address), its type (the class it belongs to), and its value.

This matters because it means Python functions can receive any kind of value as an argument, variables can be rebound to any object at any time, and features like attributes and methods are available on all values, not just class instances.

The built-in functions `id()`, `type()`, and `isinstance()` expose this model directly. `id(obj)` returns the integer identity of the object (its memory address in CPython). `type(obj)` returns the class the object belongs to. `isinstance(obj, cls)` tests whether the object is an instance of a class or any of its subclasses.

This also means Python functions are first-class objects: they have attributes, can be assigned to variables, passed as arguments, and returned from other functions. This property underpins decorators, higher-order functions, and callbacks.

Listing 1.1: Objects, identity, type, and first-class functions

```python
import inspect


# Every value has an identity, type, and value
x = 42
print(f"Value:    {x}")
print(f"Type:     {type(x)}")
print(f"Identity: {id(x)}")
print(f"Is int:   {isinstance(x, int)}")

# Functions are first-class objects
def greet(name: str) -> str:
    return f"Hello, {name}!"

print(f"\nFunction name:  {greet.__name__}")
print(f"Function type:  {type(greet)}")
print(f"Is callable:    {callable(greet)}")

# Functions can be assigned to variables
say_hello = greet
print(say_hello("Alice"))

# Functions can be passed as arguments
def apply(func, value):
    return func(value)

print(apply(str.upper, "python"))

# Functions can have attributes (they are objects)
greet.version = "1.0"
print(f"Custom attribute: {greet.version}")

# Classes are objects too
print(f"\nClass 'list' id: {id(list)}")
print(f"Class 'dict' id: {id(dict)}")
print(f"Even 'type' is an object: {type(type)}")

# Inspect shows the object model
for name, value in inspect.getmembers(42):
    if name == "__class__":
        print(f"\n42.__class__ = {value}")
        break
```

What is the difference between `is` and == in Python?

The `==` operator tests value equality: it calls the object's `__eq__` method to determine whether two objects have the same value. The `is` operator tests identity: it checks whether two variable names refer to the exact same object in memory (same `id()`).

This distinction is critical and a common source of bugs. Two objects can be equal in value but be distinct objects in memory, in which case `==` is `True` but `is` is `False`.

When to use `is`: Use `is` only to compare against singletons: `None`, `True`, and `False`. The idiomatic check is `if x is None`, not `if x == None`. This is both more readable and more correct, because a custom class could override `__eq__` to return `True` when compared against `None`.

Small integer caching: CPython caches small integers (typically -5 to 256) and interned short strings, so `is` may return `True` for these values even though it should not be relied upon. This is an implementation detail, not a language guarantee.

Listing 1.2: Identity vs equality: `is` vs ==

```python
# == tests value equality; is tests object identity

# Two equal lists that are different objects
a = [1, 2, 3]
b = [1, 2, 3]
print(f"a == b: {a == b}")    # True (same value)
print(f"a is b: {a is b}")    # False (different objects)
print(f"id(a): {id(a)}, id(b): {id(b)}")

# Assigning the same object
c = a
print(f"\na is c: {a is c}")  # True (same object)

# Singleton comparisons: always use 'is' for None
def process(value=None):
    if value is None:          # correct
        return "no value"
    if value == None:          # works, but wrong style
        return "no value"      # a class could override __eq__
    return str(value)

print(f"\n{process()}")
print(f"{process(0)}")         # 0 is falsy but not None

# Small integer caching (CPython implementation detail)
x = 256
y = 256
print(f"\n256 is 256: {x is y}")    # True (cached)

x = 257
y = 257
print(f"257 is 257: {x is y}")     # False in many contexts
```

```
                                        # (separate objects, not cached)

# String interning (another CPython detail)
s1 = "hello"
s2 = "hello"
print(f"\n'hello' is 'hello': {s1 is s2}")  # True (interned)

# Never rely on these implementation details in production code
# Always use == for value comparisons, is only for singletons
```

1.2 Mutability, Collections, and Truthiness

What is the difference between mutable and immutable objects in Python?

Mutability describes whether an object's value can change after it is created. Mutable objects can be modified in place. Immutable objects cannot; any operation that appears to modify them creates a new object instead.

Immutable types: `int`, `float`, `complex`, `bool`, `str`, `bytes`, `tuple`, `frozenset`.

Mutable types: `list`, `dict`, `set`, `bytearray`, and most user-defined class instances.

Immutability has several practical implications. Immutable objects can be used as dictionary keys and set members (they are hashable). Immutable objects are safer to share between threads without locks. When you pass an immutable value to a function, the function cannot modify the caller's variable.

A common pitfall is assuming that a tuple is always immutable. A tuple is immutable in the sense that you cannot reassign its slots, but if a slot holds a reference to a mutable object (like a list), the contents of that list can still change.

Listing 1.3: Mutable vs immutable objects and the tuple pitfall

```
# Immutable: str, int, tuple, frozenset, bytes
name = "python"
# name[0] = "P" # raises TypeError: 'str' does not support item assignment
print(f"Strings are immutable: 'name.upper()' creates a new string")
upper = name.upper()
print(f"Original: {id(name)}, New: {id(upper)}")  # different objects

# Mutable: list, dict, set
items = [1, 2, 3]
items.append(4)              # modifies in place
print(f"\nList mutated: {items}")

# Mutable default argument pitfall (classic bug)
def add_item(item, collection=[]):    # BAD: mutable default
```

```python
    collection.append(item)
    return collection

print(add_item("a"))    # ['a']
print(add_item("b"))    # ['a', 'b'] <- shared state!

def add_item_fixed(item, collection=None):  # GOOD
    if collection is None:
        collection = []
    collection.append(item)
    return collection

print(add_item_fixed("a"))  # ['a']
print(add_item_fixed("b"))  # ['b'] <- independent

# The tuple-with-mutable pitfall
point = ([0, 0], [1, 1])
# point[0] = [5, 5] # raises TypeError
point[0].append(99)    # works! the list inside is still mutable
print(f"\nTuple content mutated: {point}")

# frozenset: immutable set (hashable)
fs = frozenset([1, 2, 3])
lookup = {fs: "frozen set as key"}   # valid dictionary key
print(f"frozenset as dict key: {lookup[fs]}")

# Regular set cannot be a dict key
try:
    bad = {{1, 2}: "value"}
except TypeError as e:
    print(f"set as dict key fails: {e}")
```

What is the difference between a list and a tuple in Python?

Lists and tuples are both ordered sequences, but they differ in mutability, performance, and semantic convention.

Lists (`[]`) are mutable sequences. You can append, insert, remove, and reorder elements. They are backed by a dynamic array that resizes as needed. Use a list when you have a homogeneous collection of items that may grow or change.

Tuples (`()`) are immutable sequences. Once created, they cannot be modified. They are slightly more memory-efficient and faster to create than lists of the same size. Tuples are hashable if all their elements are hashable, so they can be used as dictionary keys.

Semantic convention: Experienced Python developers use tuples to represent heterogeneous, fixed records (like a coordinate pair `(x, y)` or a database row) and lists to represent homogeneous, variable-length collections. A function that returns multiple values actually returns a tuple.

Performance note: For large read-only sequences, tuples consume less memory than lists. The CPython constant-folding optimization can also cache tuples of constants at compile time.

Listing 1.4: Lists vs tuples: mutability, hashing, and performance

```python
import sys
import timeit

# List: mutable, dynamic, homogeneous by convention
scores = [88, 72, 95, 61, 84]
scores.append(91)
scores.sort()
print(f"Sorted scores: {scores}")

# Tuple: immutable, fixed, heterogeneous by convention
# Represents a record: (name, age, city)
person = ("Alice", 30, "New York")
name, age, city = person   # tuple unpacking
print(f"\n{name} is {age} years old in {city}")

# Tuples as dictionary keys (they are hashable)
grid = {}
grid[(0, 0)] = "origin"
grid[(1, 2)] = "point A"
print(f"\nGrid origin: {grid[(0, 0)]}")

# Lists cannot be dictionary keys
try:
    bad = {[0, 0]: "origin"}
except TypeError as e:
    print(f"List as key fails: {e}")

# Memory comparison
list_val = [1, 2, 3, 4, 5]
tuple_val = (1, 2, 3, 4, 5)
print(f"\nList  size: {sys.getsizeof(list_val)} bytes")
print(f"Tuple size: {sys.getsizeof(tuple_val)} bytes")

# Creation speed (tuple is faster for constants)
list_time = timeit.timeit("[1, 2, 3, 4, 5]", number=1_000_000)
tuple_time = timeit.timeit("(1, 2, 3, 4, 5)", number=1_000_000)
print(f"\nList  creation (1M): {list_time:.3f}s")
print(f"Tuple creation (1M): {tuple_time:.3f}s")

# Function returning multiple values returns a tuple
def min_max(values):
    return min(values), max(values)   # implicit tuple

low, high = min_max(scores)
print(f"\nMin: {low}, Max: {high}")
```

How does Python determine the truthiness of an object?

Python evaluates any object in a boolean context (inside `if`, `while`, `and`, `or`, `not`) by calling the object's `__bool__` method. If the object does not define `__bool__`, Python falls back to `__len__`:

an object is falsy if its length is zero. If neither method is defined, the object is truthy.

Built-in falsy values: `None`, `False`, zero (`0`, `0.0`, `0j`), empty sequences and collections (`""`, `[]`, `()`, `{}`, `set()`, `b""`), and `range(0)`.

Everything else is truthy by default.

Pythonic code uses this directly: prefer `if items:` over `if len(items) != 0:`, and `if value is not None:` when you need to distinguish between `None` and an empty container. Custom classes can define their own truthiness by implementing `__bool__` or `__len__`.

Listing 1.5: Truthiness, falsy values, and custom __bool__

```python
# Built-in falsy values
falsy_values = [None, False, 0, 0.0, 0j, "", [], (), {}, set(), b"", range(0)]

for val in falsy_values:
    if not val:
        print(f"Falsy: {repr(val):12} type={type(val).__name__}")

# Everything else is truthy
truthy_values = [True, 1, -1, "0", " ", [0], (None,), {"key": None}]
print()
for val in truthy_values:
    if val:
        print(f"Truthy: {repr(val):16} type={type(val).__name__}")

# Pythonic usage
items = []
# Non-pythonic:
if len(items) != 0:
    print("has items (verbose)")

# Pythonic:
if items:
    print("has items")
else:
    print("\nEmpty list is falsy -- prefer 'if items:' over 'if len(items) != 0:'")

# Distinguishing None from empty
def process(data=None):
    if data is None:             # was nothing passed at all?
        return "no argument"
    if not data:                 # was an empty container passed?
        return "empty container"
    return f"processing {len(data)} items"

print(f"\n{process()}")
print(f"{process([])}")
print(f"{process([1, 2, 3])}")

# Custom __bool__
class BankAccount:
    def __init__(self, balance: float):
        self.balance = balance
```

```python

    def __bool__(self) -> bool:
        return self.balance > 0

account_active = BankAccount(100.0)
account_empty = BankAccount(0.0)
print(f"\nActive account truthy: {bool(account_active)}")
print(f"Empty  account truthy: {bool(account_empty)}")

if account_active:
    print("Account has funds")
if not account_empty:
    print("Account is empty")
```

1.3 Scope, Strings, and Copies

What are Python's scope rules, and how do `global` and `nonlocal` work?

Python resolves names by searching four nested scopes in order, remembered by the acronym **LEGB**: **Local** (the current function), **Enclosing** (any outer function scopes, for closures), **Global** (the module's top-level namespace), and **Built-in** (Python's built-in names like `len`, `print`, `range`).

When a name is *assigned* inside a function, Python treats it as a local variable for the entire function body — even lines before the assignment. Reading the name before its assignment raises `UnboundLocalError`. This surprises developers who expect assignment inside a function to fall through to the global scope.

`global`**:** Declares that a name refers to the module-level variable. Assignments inside the function then modify the global, not a local. Avoid relying on `global` in production code; it creates hidden coupling between functions.

`nonlocal`**:** Declares that a name refers to the nearest enclosing scope (not global). Required when a nested function needs to *rebind* (not just read) a variable from an outer function. The classic use case is a counter inside a closure.

Listing 1.6: LEGB scope, global, nonlocal, and UnboundLocalError

```python
# LEGB scope: Local -> Enclosing -> Global -> Built-in
# Python resolves names by searching these four scopes in order.

# Global scope
x = "global"


def outer():
    x = "enclosing"      # Enclosing scope for inner()
```

```python

    def inner():
        x = "local"      # Local scope
        print(f"inner sees: {x}")     # local

    inner()
    print(f"outer sees: {x}")         # enclosing


outer()
print(f"module sees: {x}")            # global


# — UnboundLocalError pitfall —
count = 0

def broken_increment():
    # Because count is assigned later in this function body, Python
    # treats it as local for the ENTIRE function — even before the assignment.
    # Reading it before assignment raises UnboundLocalError.
    try:
        print(count)        # <- UnboundLocalError here
    except UnboundLocalError as e:
        print(f"UnboundLocalError: {e}")
    count = 1                # this makes 'count' local to the whole function

broken_increment()


# — global keyword —
total = 0

def add_to_total(n: int) -> None:
    global total           # declare we want the module-level 'total'
    total += n

add_to_total(5)
add_to_total(3)
print(f"total: {total}")  # 8
# Avoid global in production — prefer returning values or using a class


# — nonlocal: rebind in enclosing scope —
def make_counter(start: int = 0):
    count = start

    def increment(by: int = 1) -> int:
        nonlocal count      # rebind the enclosing variable, not a new local
        count += by
        return count

    def reset() -> None:
        nonlocal count
        count = start

    return increment, reset


inc, rst = make_counter(10)
```

```
print(inc())     # 11
print(inc(5))    # 16
rst()
print(inc())     # 11 - reset worked
```

How does string formatting work in Python, and when should you use f-strings?

Python offers four string formatting approaches, each with different trade-offs:

- **% formatting (legacy):** C-style, e.g. `"%s is %d" % (name, age)`. Still works in Python 3 but is considered outdated. Use only when maintaining old codebases or interfacing with C extensions that expect this format.
- `str.format()` **(Python 2.6+):** Named placeholders, positional arguments, format specifications. More readable than % formatting and supports reuse of arguments. Still useful for template strings built at runtime.
- **f-strings (Python 3.6+):** Inline expressions inside string literals, evaluated at definition time. The fastest, most readable, and most commonly used approach in modern code.
- `string.Template`**:** Simple `$name` substitution, safe for user-provided templates because it does not evaluate arbitrary expressions. Useful in security-conscious contexts.

Performance note: f-strings are slightly faster than `str.format()` or % formatting because they compile to optimized bytecode. For hot-path logging, prefer lazy formatting: pass `%s` arguments to the logger rather than formatting the string eagerly, so no string is built if the log level is inactive.

Listing 1.7: String formatting: %, format(), f-strings, and Template

```
# String formatting in Python: %, str.format(), f-strings, string.Template.
import string
import logging

name = "Alice"
score = 98.5
rank = 1

# 1. % formatting (legacy, Python 2 style - avoid in new code)
msg_pct = "%-10s scored %.1f (rank %d)" % (name, score, rank)
print(f"%%: {msg_pct}")

# 2. str.format() - named placeholders, reusable template strings
template = "{name:<10} scored {score:.1f} (rank {rank})"
msg_fmt = template.format(name=name, score=score, rank=rank)
print(f"format(): {msg_fmt}")

# Positional with reuse
```

```
msg_reuse = "{0} vs {0}: same argument twice".format("echo")
print(msg_reuse)

# 3. f-strings (Python 3.6+) - fastest, most readable
msg_fstr = f"{name:<10} scored {score:.1f} (rank {rank})"
print(f"f-string: {msg_fstr}")

# f-strings can contain arbitrary expressions
import math
print(f"pi = {math.pi:.4f}")
print(f"2^10 = {2**10}")
print(f"{name!r}")            # calls repr(): 'Alice'
print(f"{name!s}")            # calls str(): Alice

# Python 3.8+ self-documenting expression
x = 42
print(f"{x=}")                 # x=42

# 4. string.Template - safe for user-provided patterns
t = string.Template("Hello, $name! Your score is $score.")
print(t.substitute(name=name, score=score))
# safe_substitute() leaves unresolved placeholders intact instead of raising
partial_template = string.Template("Hello, $name! Balance: $balance")
print(partial_template.safe_substitute(name=name))  # $balance left as-is


# PERFORMANCE: lazy logging - do NOT format eagerly
logger = logging.getLogger(__name__)
# BAD: logger.debug(f"User {name} processed") # string built even if DEBUG disabled
# GOOD: logger.debug("User %s processed", name) # built only when the level is active
```

How does slicing work in Python, and what is extended slicing?

Slicing returns a new object containing a portion of a sequence. The basic form is `seq[start:stop:step]`, where all three arguments are optional and default to `None` (interpreted as start-of-sequence, end-of-sequence, and step of 1 respectively). Negative indices count from the end; a negative step reverses direction.

Important: Slicing a list, string, or tuple always creates a *new* object, so modifying the slice does not affect the original. Slicing a NumPy array, by contrast, returns a *view* — a common source of bugs when working with data science code.

Extended slicing (step argument) allows selecting every N-th element, reversing a sequence with `[::-1]`, or striding through binary data. Custom objects can support slicing by implementing `__getitem__` with a `slice` object argument.

Listing 1.8: Slicing, extended slicing, negative indices, and slice objects

```
# Slicing in Python: seq[start:stop:step]
# All three arguments are optional; negative indices count from the end.

```

```python
data = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Basic slices
print(data[2:5])       # [2, 3, 4] — start inclusive, stop exclusive
print(data[:3])        # [0, 1, 2] — omit start -> from beginning
print(data[7:])        # [7, 8, 9] — omit stop -> to end
print(data[:])         # full shallow copy

# Negative indices
print(data[-3:])       # [7, 8, 9] — last 3 elements
print(data[:-2])       # [0..7] — all except last 2
print(data[-5:-2])     # [5, 6, 7]

# Step (extended slicing)
print(data[::2])       # [0, 2, 4, 6, 8] — every other element
print(data[1::2])      # [1, 3, 5, 7, 9] — odd indices
print(data[::-1])      # [9..0] — reverse (idiomatic)
print(data[8:1:-2])    # [8, 6, 4, 2] — start=8, stop=1 (excl), step=-2

# Slicing creates a NEW object for lists, strings, tuples
a = [1, [2, 3], 4]
b = a[:]                # shallow copy
b[0] = 99
b[1].append(9)          # inner list is shared!
print(f"a: {a}")        # a[0]=1 unchanged, a[1]=[2, 3, 9] mutated
print(f"b: {b}")        # b[0]=99

# Slice assignment modifies in place
nums = list(range(10))
nums[2:5] = [20, 30]    # replace 3 elements with 2
print(nums)              # [0, 1, 20, 30, 5, 6, 7, 8, 9]

nums[::2] = [0] * 5     # replace every other element
print(nums)

# slice() objects — useful for named, reusable slices
HEADER = slice(None, 4)
TAIL   = slice(-3, None)
text = "Hello, World!"
print(text[HEADER])     # 'Hell'
print(text[TAIL])       # 'd!'... actually last 3 chars

# Custom __getitem__ receives a slice object
class LabelledList:
    def __init__(self, data): self._data = data
    def __getitem__(self, key):
        if isinstance(key, slice):
            return LabelledList(self._data[key])
        return self._data[key]
    def __repr__(self): return f"LabelledList({self._data})"

ll = LabelledList([10, 20, 30, 40, 50])
print(ll[1:4])
```

What is the difference between a shallow copy and a deep copy in Python?

Both copy mechanisms create a new container object, but they differ in how nested objects are handled.

A **shallow copy** creates a new container but inserts *references* to the same objects that the original container held. For flat containers (no nested mutables), a shallow copy is effectively independent. For nested structures (lists of lists, dicts of dicts), mutations to the inner objects are visible through both the original and the copy.

A **deep copy** recursively copies every object encountered. The result is fully independent from the original. Deep copies are slower and consume more memory, but they are necessary when the original structure has nested mutables that must not be shared.

When to use which:

- Shallow copy: simple flat structures, when inner objects are immutable, or when sharing inner objects is intentional
- Deep copy: nested mutable data structures that need true independence
- Avoid both: for configuration objects and complex graphs, prefer constructing new instances explicitly rather than copying, to avoid surprising behavior with shared state, locks, or file handles

Listing 1.9: Shallow copy vs deep copy: behaviour with nested structures

```python
# Shallow copy vs deep copy.
# copy.copy() creates a new container with references to the same inner objects.
# copy.deepcopy() recursively copies everything.
import copy


# Flat structure: shallow copy is sufficient
flat_original = [1, 2, 3, 4]
flat_copy = copy.copy(flat_original)
flat_copy.append(99)
print(f"original: {flat_original}")  # unchanged - [1, 2, 3, 4]
print(f"copy:     {flat_copy}")       # [1, 2, 3, 4, 99]


# Nested structure: shallow copy shares inner objects
nested_original = [[1, 2], [3, 4], [5, 6]]
shallow = copy.copy(nested_original)

shallow[0].append(99)    # mutates the shared inner list
shallow.append([7, 8])   # only affects the shallow copy's outer list

print(f"\noriginal: {nested_original}")  # [[1, 2, 99], [3, 4], [5, 6]] - inner mutated!
```

```python
print(f"shallow:  {shallow}")             # [[1, 2, 99], [3, 4], [5, 6], [7, 8]]
print(f"same inner: {nested_original[0] is shallow[0]}")  # True


# Deep copy: fully independent
nested_original2 = [[1, 2], [3, 4], [5, 6]]
deep = copy.deepcopy(nested_original2)

deep[0].append(99)

print(f"\noriginal: {nested_original2}")  # [[1, 2], [3, 4], [5, 6]] - untouched
print(f"deep:     {deep}")                 # [[1, 2, 99], [3, 4], [5, 6]]
print(f"same inner: {nested_original2[0] is deep[0]}")  # False


# Shallow copy alternatives
import_list = [10, 20, 30]
c1 = import_list.copy()       # list.copy() - shallow
c2 = import_list[:]           # slice copy - shallow
c3 = list(import_list)        # constructor copy - shallow
print(f"\nAll shallow: {c1 == c2 == c3}")


# Deep copy respects __deepcopy__ and handles cycles
class Node:
    def __init__(self, val, children=None):
        self.val = val
        self.children = children or []

root = Node(1, [Node(2), Node(3, [Node(4)])])
root_copy = copy.deepcopy(root)
root_copy.children[0].val = 99
print(f"\noriginal root.children[0].val: {root.children[0].val}")  # 2, unchanged
print(f"deep copy  root.children[0].val: {root_copy.children[0].val}")  # 99
```

How do Python dictionaries work internally, and what dict methods are most important?

Python's `dict` is backed by a hash table. When you insert a key, Python calls `hash(key)` to compute an integer and uses it to find a slot in the underlying array. Lookup and insertion are O(1) average case. Worst-case O(n) occurs when many keys collide — rare with well-distributed keys, but relevant in security contexts (hash flooding attacks, mitigated in Python 3 by random hash seeds via `PYTHONHASHSEED`).

Key guarantees since Python 3.7: dictionaries maintain *insertion order*. This is part of the language specification, not just a CPython implementation detail.

Commonly tested dict methods:

- `get(key, default)`: safe lookup; returns default instead of raising `KeyError`

- `setdefault(key, default)`: insert and return default if key is absent; useful for accumulating grouped data
- `update(other)`: merges another dict or iterable of pairs in place
- `items()`, `keys()`, `values()`: return live view objects, not lists
- `pop(key, default)`: remove and return, optionally with a default
- `|` and `|=` (Python 3.9+): merge operator and in-place merge

Listing 1.10: Dict internals, ordering, common methods, and the merge operator

```python
# Dictionary internals, insertion order, and the most important dict methods.

# Insertion order preserved since Python 3.7 (language guarantee)
config = {}
config["host"] = "localhost"
config["port"] = 5432
config["db"] = "myapp"
print(f"Insertion order: {list(config.keys())}")  # ['host', 'port', 'db']


# — get(key, default): safe lookup without KeyError —
port = config.get("port", 3306)    # 5432 (key exists)
timeout = config.get("timeout", 30)  # 30 (key missing — returns default)
print(f"port={port}, timeout={timeout}")


# — setdefault: insert if missing, always return the value —
registry = {}
registry.setdefault("tags", []).append("python")
registry.setdefault("tags", []).append("django")  # key exists — appends to same list
print(f"tags: {registry['tags']}")    # ['python', 'django']

# Equivalent pattern with defaultdict:
from collections import defaultdict
dd = defaultdict(list)
dd["tags"].append("python")
dd["tags"].append("django")


# — update: merge another dict or iterable of pairs —
defaults = {"debug": False, "timeout": 30, "max_retries": 3}
overrides = {"timeout": 60, "log_level": "INFO"}
merged = {**defaults, **overrides}    # unpack merge (Python 3.5+)
print(f"merged: {merged}")

defaults.update(overrides)            # in-place
print(f"updated: {defaults}")


# — | merge operator (Python 3.9+) —
a = {"x": 1, "y": 2}
b = {"y": 99, "z": 3}
c = a | b            # new dict; b's values win on conflict
```

```python
print(f"a | b: {c}")

a |= b                # in-place merge
print(f"a |= b: {a}")


# — pop, popitem, clear —
d = {"a": 1, "b": 2, "c": 3}
val = d.pop("b", None)    # removes 'b', returns 1; default avoids KeyError
print(f"popped b={val}, remaining: {d}")

last_key, last_val = d.popitem()    # LIFO: removes last inserted item
print(f"popitem: {last_key}={last_val}, remaining: {d}")


# — items/keys/values are VIEWS, not lists —
d = {"x": 10, "y": 20}
keys_view = d.keys()
print(f"before: {keys_view}")    # dict_keys(['x', 'y'])
d["z"] = 30
print(f"after:  {keys_view}")    # dict_keys(['x', 'y', 'z']) — live view updated


# — dict comprehension —
squares = {n: n**2 for n in range(1, 6)}
filtered = {k: v for k, v in squares.items() if v > 4}
print(f"squares: {squares}")
print(f"filtered: {filtered}")
```

1.4 Sets, Numbers, and Unpacking

What is the difference between a set and a frozenset in Python?

A `set` is a mutable, unordered collection of unique hashable objects. Because it is mutable, a `set` is itself unhashable and cannot be used as a dictionary key or as a member of another set. A `frozenset` is the immutable counterpart: once created it cannot be changed, and because it is immutable it is hashable.

Set operations: Python supports the full suite of mathematical set operations via operators and their in-place equivalents:

- `a | b` — union (all elements from both); in-place: `a |= b`
- `a & b` — intersection (elements in both); in-place: `a &= b`
- `a - b` — difference (in `a` but not `b`); in-place: `a -= b`
- `a ^ b` — symmetric difference (in one but not both); in-place: `a ^= b`

Membership testing: `x in s` is O(1) average for a set because Python computes the hash of `x` and jumps directly to the relevant bucket. The equivalent test on a list is O(n) because Python scans every element until it finds a match.

frozenset use cases: Because a `frozenset` is hashable, it can serve as a dictionary key or as an element inside another set. Practical applications include immutable tag collections on a cached object, using a set of feature flags as a cache key, and constructing a set-of-sets for graph or combinatorics work.

`discard()` **vs** `remove()`**:** both remove an element from a set. `set.discard(x)` silently does nothing if `x` is not present. `set.remove(x)` raises `KeyError` if `x` is not present. Use `discard()` when absence is an acceptable condition; use `remove()` when absence indicates a programming error that should be surfaced immediately.

Listing 1.11: Set operations, frozenset as dict key and set member, discard vs remove

```python
# 11_sets_frozensets.py - set operations, frozenset as dict key, discard vs remove

# — Basic set creation —
fruits = {"apple", "banana", "cherry"}
veggies = {"carrot", "banana", "spinach"}

# Set operations using operators
print(fruits | veggies)   # union: all elements from both
print(fruits & veggies)   # intersection: elements in both
print(fruits - veggies)   # difference: in fruits but not veggies
print(fruits ^ veggies)   # symmetric difference: in one but not both

# In-place equivalents
fruits |= {"mango"}         # equivalent to fruits.update({"mango"})
fruits &= {"apple", "mango", "cherry"}

# — Membership test is O(1) average (hash-based) —
big_set = set(range(1_000_000))
print(999_999 in big_set)  # True, O(1) - unlike list O(n)

# — discard vs remove —
s = {1, 2, 3}
s.discard(99)   # no error when element is missing
try:
    s.remove(99)   # raises KeyError if missing
except KeyError as e:
    print(f"remove() raised KeyError: {e}")

# — frozenset: immutable and hashable —
tags_a = frozenset({"python", "django", "api"})
tags_b = frozenset({"python", "flask"})

# frozenset supports read-only set operations
print(tags_a & tags_b)   # {'python'}

# Can be used as a dict key (regular set cannot)
```

```python
cache = {tags_a: "result_a", tags_b: "result_b"}
print(cache[frozenset({"python", "django", "api"})])

# Can be a member of another set
set_of_sets = {frozenset({"x", "y"}), frozenset({"a", "b"})}
print(set_of_sets)

# — Real use case: permission set as cache key —
def get_allowed_views(permissions: frozenset) -> list:
    # Cache or look up allowed views for an immutable permission set.
    _cache = {
        frozenset({"read"}): ["list", "detail"],
        frozenset({"read", "write"}): ["list", "detail", "create", "update"],
    }
    return _cache.get(permissions, [])

perms = frozenset({"read", "write"})
print(get_allowed_views(perms))
```

What is the walrus operator (:=) and when should you avoid it?

The walrus operator `:=`, introduced in Python 3.8 (PEP 572), is an *assignment expression*. It assigns a value to a name and simultaneously evaluates to that value within the surrounding expression. This differs from a regular assignment statement, which assigns but returns nothing.

Primary idioms: the two most common uses are chunked file reading and regex matching:

- `while chunk := f.read(8192):` — reads a chunk, assigns it, and tests truthiness in one step. The loop exits as soon as `read()` returns an empty bytes object.
- `if m := re.match(pattern, text):` — performs the match, assigns the result, and enters the block only when the match succeeded.

List comprehensions: the walrus operator prevents computing a transformation twice. The expression `[y for x in data if (y := transform(x)) > 0]` calls `transform(x)` once per item, uses the result in the filter condition, and emits the same result in the output — no re-computation required.

When to avoid it: deeply nested walrus expressions — where the assigned name is used inside another walrus expression or complex boolean logic — reduce readability significantly. If the expression is short and a regular assignment before the condition is equally concise, prefer the assignment. PEP 572 itself cautions that the operator should not be used to replace every two-line assignment-then-test pattern, only the cases where the improvement is clear.

Syntax restriction: the walrus operator cannot appear at the statement level. Writing `x := 5` as a standalone statement is a `SyntaxError`; it must appear inside an expression context such as

a parenthesised expression, a comprehension, or a conditional.

Listing 1.12: Walrus operator: while chunk loop, regex match, and list comprehension

```python
# 12_walrus_operator.py — walrus operator (:=) assignment expressions

import re
import io

# — Pattern 1: while loop with file reading —
# Without walrus: must read twice or use a sentinel
data = io.BytesIO(b"Hello, world! This is a test of chunked reading.")

chunks = []
while chunk := data.read(8):    # assign and test in one expression
    chunks.append(chunk)
print(b"".join(chunks))

# — Pattern 2: regex match with early return —
text = "Order #12345 placed on 2024-03-15"
pattern = r"Order #(\d+)"

if m := re.search(pattern, text):
    print(f"Found order number: {m.group(1)}")
else:
    print("No order found")

# Without walrus: less clean
match = re.search(pattern, text)
if match:
    print(match.group(1))

# — Pattern 3: list comprehension — compute once, use in filter and output —
def transform(x: int) -> int:
    # Expensive transformation we only want to call once.
    return x * x - 3

data_list = range(-5, 6)

# With walrus: transform is called once per item
results = [y for x in data_list if (y := transform(x)) > 0]
print(results)

# Without walrus: transform called twice (inefficient)
results_old = [transform(x) for x in data_list if transform(x) > 0]

# — When to AVOID the walrus operator —
# Bad: deeply nested walrus reduces readability
# if (a := (b := f(x)) + g(b)) > 0: ... # confusing, avoid

# Bad: using walrus at statement level is a SyntaxError
# x := 5 # SyntaxError — must be inside an expression
(x := 5)  # valid — walrus inside a parenthesised expression
print(x)

# Good rule: if a regular assignment is just as short, prefer it for clarity
```

How does iterable unpacking and starred assignment work in Python?

Python's unpacking syntax binds multiple names on the left side of an assignment to the elements of any iterable on the right. The iterable does not need to be a tuple; a list, string, generator, or any other iterable works.

Basic unpacking: `a, b = (1, 2)` assigns `1` to `a` and `2` to `b`. The number of names must match the number of elements, or a `ValueError` is raised.

Extended (starred) unpacking: a single `*` prefix on one name collects all remaining elements into a list. `first, *rest = [1, 2, 3, 4]` gives `first = 1` and `rest = [2, 3, 4]`. The starred name can appear anywhere: `*head, last = data` collects everything except the final element. The `*_` idiom discards middle elements while capturing only the first and last: `first, *_, last = data`.

Nested unpacking: `(a, b), c = (1, 2), 3` destructures a nested iterable in one assignment. This is particularly useful when iterating over structured data: `for (x, y) in list_of_pairs:`.

Variable swap: `a, b = b, a` swaps two variables without a temporary. Python fully evaluates the right-hand side as a tuple before performing any assignment, so both original values are preserved during the swap.

Practical patterns: `head, *tail = some_list` splits a sequence for recursive processing; unpacking a function return value into named variables documents intent; destructuring loop tuples (`for name, value in config.items()`) improves readability. The `*args` and `**kwargs` syntax in function signatures applies the same mechanism to argument collections (covered in depth in Chapter 3).

Listing 1.13: Extended unpacking, nested unpacking, swap, starred discard, and loop destructuring

```python
# 13_iterable_unpacking.py - iterable unpacking and starred assignment

# - Basic unpacking: works with any iterable -
a, b = (1, 2)            # tuple
x, y = [10, 20]          # list
p, q = "hi"              # string (iterable of chars)
print(a, b, x, y, p, q)

# - Extended (starred) unpacking -
first, *rest = [1, 2, 3, 4, 5]
print(first)   # 1
print(rest)    # [2, 3, 4, 5] - rest is always a list

*head, last = [1, 2, 3, 4, 5]
print(head, last)   # [1, 2, 3, 4], 5
```

```python

# Practical: split head from tail of a list
some_list = [10, 20, 30, 40]
head_item, *tail = some_list
print(head_item, tail)  # 10, [20, 30, 40]

# — Discarding middle elements with *_ —
data = ["Alice", "ignored_1", "ignored_2", "active"]
first_item, *_, last_item = data
print(first_item, last_item)  # Alice active

# — Nested unpacking —
(a2, b2), c2 = (1, 2), 3
print(a2, b2, c2)   # 1 2 3

# Works with loop iteration over structured data
points = [(1, 2), (3, 4), (5, 6)]
for x_coord, y_coord in points:
    print(f"x={x_coord}, y={y_coord}")

# — Swap without a temporary variable —
# Right-hand side is fully evaluated as a tuple before assignment
m, n = 10, 20
m, n = n, m
print(m, n)   # 20 10

# — Destructuring structured returns —
def get_bounds(values):
    return min(values), max(values)

low, high = get_bounds([3, 1, 4, 1, 5, 9, 2])
print(low, high)  # 1 9

# — Brief: *args and **kwargs in function signatures —
def summarise(*args, **kwargs):
    # args is a tuple; kwargs is a dict.
    print(f"positional: {args}")
    print(f"keyword: {kwargs}")

summarise(1, 2, 3, name="Alice", active=True)
```

What numeric types does Python provide, and when do you need Decimal or Fraction?

Python provides four built-in numeric types and two standard-library types for specialised arithmetic.

`int`: arbitrary-precision integer. Python automatically promotes to a bigint representation when the value exceeds the platform word size, so integer overflow is impossible. All arithmetic on `int` values is exact.

`float`: IEEE 754 double-precision floating-point. Provides 15–17 significant decimal digits of

precision. The canonical pitfall is `0.1 + 0.2 != 0.3`: both `0.1` and `0.2` cannot be represented exactly in binary floating-point, so their sum accumulates a small error. Never compare floats with `==`; use `math.isclose(a, b, rel_tol=1e-9)` instead.

`complex`: complex number written as `3+4j`. `abs(z)` returns the magnitude. Rarely used in application code but available for scientific or signal-processing work.

`decimal.Decimal`: arbitrary-precision decimal arithmetic. Numbers are stored as decimal fractions, not binary fractions, so `Decimal("0.1") + Decimal("0.2") == Decimal("0.3")` is `True`. Precision is configurable via `decimal.getcontext().prec`. Essential for financial calculations, currency, and tax, where rounding errors in binary floating-point are unacceptable.

`fractions.Fraction`: exact rational arithmetic. `Fraction(1, 3) + Fraction(1, 6) == Fraction(1, 2)` is `True` because Python keeps the exact numerator and denominator throughout the calculation. Useful for geometry, probability, and any domain where exact division matters.

Numeric tower: Python defines abstract base classes in the `numbers` module that form a hierarchy: `numbers.Number` > `numbers.Complex` > `numbers.Real` > `numbers.Rational` > `numbers.Integral`. These allow code to accept "any real number" or "any integer" without coupling to a specific concrete type.

Listing 1.14: int, float, Decimal, Fraction, math.isclose, and the numeric tower ABCs

```python
# 14_numeric_types.py — int, float, Decimal, Fraction, and the numeric tower

import math
import numbers
from decimal import Decimal, getcontext
from fractions import Fraction

# — int: arbitrary precision, no overflow —
big = 2 ** 1000    # Python handles this natively — no overflow
print(type(big), str(big)[:30], "...")

# — float: IEEE 754 double precision, ~15-17 significant digits —
print(0.1 + 0.2)            # 0.30000000000000004
print(0.1 + 0.2 == 0.3)     # False — canonical floating-point pitfall
print(repr(0.1))             # shows the actual stored approximation

# Correct float comparison: use math.isclose
print(math.isclose(0.1 + 0.2, 0.3, rel_tol=1e-9))  # True

# — complex: available built-in —
z = 3 + 4j
print(abs(z))          # 5.0 — magnitude via Pythagoras
print(z.real, z.imag)

# — decimal.Decimal: arbitrary-precision decimal arithmetic —
```

```python
getcontext().prec = 28    # configure precision globally

d1 = Decimal("0.1")
d2 = Decimal("0.2")
print(d1 + d2)               # Decimal('0.3') - exact
print(d1 + d2 == Decimal("0.3"))   # True

# Essential for financial calculations
price = Decimal("19.99")
tax_rate = Decimal("0.0825")
tax = (price * tax_rate).quantize(Decimal("0.01"))
print(f"Tax: ${tax}")    # Tax: $1.65

# - fractions.Fraction: exact rational arithmetic -
f1 = Fraction(1, 3)
f2 = Fraction(1, 6)
print(f1 + f2)                     # Fraction(1, 2)
print(f1 + f2 == Fraction(1, 2))   # True - exact
print(Fraction(0.1))              # shows the actual float fraction

# - Python's numeric tower (abstract base classes) -
# numbers.Number > Complex > Real > Rational > Integral
for value in [1, 1.0, 1+0j, Fraction(1, 2), Decimal("1")]:
    is_integral = isinstance(value, numbers.Integral)
    is_real = isinstance(value, numbers.Real)
    print(f"{repr(value):20s}  Integral={is_integral}  Real={is_real}")
```

What does *hashable* mean in Python, and why does it matter?

An object is hashable if it implements a `__hash__` method that returns a stable integer for the lifetime of the object and is *consistent with equality*: if `a == b` then `hash(a) == hash(b)`. Python uses this hash to locate the bucket in a set or dictionary hash table, making `x in s` O(1) average for both.

Built-in hashability rules:

- All built-in *immutable* types (`int`, `float`, `str`, `bytes`, `bool`, `frozenset`) are hashable.
- All built-in *mutable* containers (`list`, `dict`, `set`) are **not** hashable.
- A `tuple` is hashable only if every element it contains is itself hashable. `hash((1, 2))` works; `hash((1, [2]))` raises `TypeError`.

Custom classes: by default, user-defined classes inherit `__hash__` from `object` (identity-based). As soon as you define `__eq__`, Python implicitly sets `__hash__ = None`, making instances unhashable — because the hash contract would be broken if two equal objects could have different hashes. To restore hashability, you must define `__hash__` explicitly, typically by delegating to a tuple of the fields used in `__eq__`.

Practical consequences:

- `@dataclass(frozen=True)` generates both `__eq__` and `__hash__` based on all fields, making frozen dataclasses safe as dict keys and set members.
- Never include mutable attributes in `__hash__`: if the object changes, its hash changes, and it becomes unfindable in any set or dict it was already inserted into.
- `hash()` is not guaranteed to be the same across Python processes (random seed via `PYTHONHASHSEED`). Do not persist or compare hash values across processes.

Listing 1.15: Hashability: built-in rules, the __eq__/__hash__ contract, custom implementation, and frozen dataclass

```python
# - Hashability in Python ----------------

# 1. Hash basics: built-in immutable types are hashable
print(hash(42))             # stable integer
print(hash("hello"))        # stable for the process lifetime (PYTHONHASHSEED)
print(hash((1, 2, 3)))      # tuple is hashable when all elements are hashable

# Mutable built-ins are NOT hashable
try:
    hash([1, 2, 3])         # list
except TypeError as e:
    print(e)                # unhashable type: 'list'

try:
    hash({"a": 1})          # dict
except TypeError as e:
    print(e)                # unhashable type: 'dict'


# 2. Tuple is only hashable if every element is hashable
print(hash((1, "hello", (2, 3))))    # OK - all elements hashable

try:
    hash((1, [2, 3]))       # contains a list -> not hashable
except TypeError as e:
    print(e)


# 3. Custom class: defining __eq__ sets __hash__ to None by default
class BadPoint:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __eq__(self, other):
        return (self.x, self.y) == (other.x, other.y)
    # __hash__ is implicitly None now -> unhashable


try:
    s = {BadPoint(1, 2)}
except TypeError as e:
    print(e)                # unhashable type: 'BadPoint'


# 4. Correctly implement both __eq__ and __hash__ together
```

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

    def __eq__(self, other):
        return isinstance(other, Point) and (self.x, self.y) == (other.x, other.y)

    def __hash__(self):
        return hash((self.x, self.y))   # delegate to tuple hash


p1, p2 = Point(1, 2), Point(1, 2)
print(p1 == p2)             # True
print(p1 is p2)             # False
print(hash(p1) == hash(p2))# True

point_set = {p1, p2}
print(len(point_set))       # 1 - deduplicated via __eq__ after hash match


# 5. Frozen dataclass generates __hash__ automatically
from dataclasses import dataclass

@dataclass(frozen=True)
class Vector:
    x: float
    y: float

v = Vector(3.0, 4.0)
print(hash(v))              # works because frozen=True implies immutability
cache = {v: "magnitude_5"}
```