Effective Software Development Series
Scott Meyers, Consulting Editor

Effective Python

125 Specific Ways to Write Better Python

Third Edition

Brett Slatkin

Effective Python

Third Edition

Effective Python: 125 Specific Ways to Write Better Python

Table of Contents

Cover

Half Title Page

Title Page

Copyright Page

Contents at a Glance

Contents

Preface

Acknowledgments

About the Author

Chapter 1 Pythonic Thinking

- Item 1: Know Which Version of Python Youre Using
- Item 2: Follow the PEP 8 Style Guide
- Item 3: Never Expect Python to Detect Errors at Compile Time
- Item 4: Write Helper Functions Instead of Complex Expressions
- Item 5: Prefer Multiple-Assignment Unpacking over Indexing
- Item 6: Always Surround Single-Element Tuples with Parentheses
- Item 7: Consider Conditional Expressions for Simple Inline Logic
- Item 8: Prevent Repetition with Assignment Expressions
- Item 9: Consider match for Destructuring in Flow Control; Avoid When if Statements Are Sufficient

Chapter 2 Strings and Slicing

Pearson

Table of Contents

Item 10: Know the Differences Between bytes and str

Item 11: Prefer Interpolated F-Strings over C-Style Format Strings and str.format

Item 12: Understand the Difference Between repr and str when Printing Objects

Item 13: Prefer Explicit String Concatenation over Implicit, Especially in Lists

Item 14: Know How to Slice Sequences

Item 15: Avoid Striding and Slicing in a Single Expression

Item 16: Prefer Catch-All Unpacking over Slicing

Chapter 3 Loops and Iterators

Item 17: Prefer enumerate over range

Item 18: Use zip to Process Iterators in Parallel

Item 19: Avoid else Blocks After for and while Loops

Item 20: Never Use for Loop Variables After the Loop Ends

Item 21: Be Defensive when Iterating over Arguments

Item 22: Never Modify Containers While Iterating over Them; Use Copies or Caches Instead

Item 23: Pass Iterators to any and all for Efficient Short-Circuiting Logic

Item 24: Consider itertools for Working with Iterators and Generators

Chapter 4 Dictionaries

Item 25: Be Cautious when Relying on Dictionary Insertion Ordering

Item 26: Prefer get over in and KeyError to Handle Missing Dictionary Keys

Item 27: Prefer defaultdict over setdefault to Handle Missing Items in Internal State

Pearson

Table of Contents

Item 28: Know How to Construct Key-Dependent Default Values with __missing__

Item 29: Compose Classes Instead of Deeply Nesting Dictionaries, Lists, and Tuples

Chapter 5 Functions

Item 30: Know That Function Arguments Can Be Mutated

Item 31: Return Dedicated Result Objects Instead of Requiring Function Callers to Unpack More Than Three Variables

Item 32: Prefer Raising Exceptions to Returning None

Item 33: Know How Closures Interact with Variable Scope and nonlocal

Item 34: Reduce Visual Noise with Variable Positional Arguments

Item 35: Provide Optional Behavior with Keyword Arguments

Item 36: Use None and Docstrings to Specify Dynamic Default Arguments

Item 37: Enforce Clarity with Keyword-Only and Positional-Only Arguments

Item 38: Define Function Decorators with functools.wraps

Item 39: Prefer functools.partial over lambda Expressions for Glue Functions

Chapter 6 Comprehensions and Generators

Item 40: Use Comprehensions Instead of map and filter

Item 41: Avoid More Than Two Control Subexpressions in Comprehensions

Item 42: Reduce Repetition in Comprehensions with Assignment Expressions

Item 43: Consider Generators Instead of Returning Lists

Pearson

Table of Contents

Item 44: Consider Generator Expressions for Large List Comprehensions

Item 45: Compose Multiple Generators with yield from

Item 46: Pass Iterators into Generators as Arguments Instead of Calling the send Method

Item 47: Manage Iterative State Transitions with a Class Instead of the Generator throw Method

Chapter 7 Classes and Interfaces

Item 48: Accept Functions Instead of Classes for Simple Interfaces

Item 49: Prefer Object-Oriented Polymorphism over Functions with isinstance Checks

Item 50: Consider functools.singledispatch for Functional-Style Programming Instead of Object-Oriented Polymorphism

Item 51: Prefer dataclasses for Defining Lightweight Classes

Item 52: Use @classmethod Polymorphism to Construct Objects Generically

Item 53: Initialize Parent Classes with super

Item 54: Consider Composing Functionality with Mix-in Classes

Item 55: Prefer Public Attributes over Private Ones

Item 56: Prefer dataclasses for Creating Immutable Objects

Item 57: Inherit from collections.abc Classes for Custom Container Types

Chapter 8 Metaclasses and Attributes

Item 58: Use Plain Attributes Instead of Setter and Getter Methods

Item 59: Consider @property Instead of Refactoring Attributes

Item 60: Use Descriptors for Reusable @property Methods

Item 61: Use __getattr__, __getattribute__, and __setattr__ for Lazy Attributes

Table of Contents

Item 62: Validate Subclasses with __init_subclass__
Item 63: Register Class Existence with __init_subclass__
Item 64: Annotate Class Attributes with __set_name__
Item 65: Consider Class Body Definition Order to Establish Relationships Between Attributes
Item 66: Prefer Class Decorators over Metaclasses for Composable Class Extensions
Chapter 9 Concurrency and Parallelism
Item 67: Use subprocess to Manage Child Processes
Item 68: Use Threads for Blocking I/O; Avoid for Parallelism
Item 69: Use Lock to Prevent Data Races in Threads
Item 70: Use Queue to Coordinate Work Between Threads
Item 71: Know How to Recognize When Concurrency Is Necessary
Item 72: Avoid Creating New Thread Instances for On-Demand Fan-out
Item 73: Understand How Using Queue for Concurrency Requires Refactoring
Item 74: Consider ThreadPoolExecutor When Threads Are Necessary for Concurrency
Item 75: Achieve Highly Concurrent I/O with Coroutines
Item 76: Know How to Port Threaded I/O to asyncio
Item 77: Mix Threads and Coroutines to Ease the Transition to asyncio
Item 78: Maximize Responsiveness of asyncio Event Loops with async-Friendly Worker Threads
Item 79: Consider concurrent.futures for True Parallelism
Chapter 10 Robustness
Item 80: Take Advantage of Each Block in try/except/else/finally

Table of Contents

Item 81: assert Internal Assumptions and raise Missed Expectations

Item 82: Consider contextlib and with Statements for Reusable try/finally Behavior

Item 83: Always Make try Blocks as Short as Possible

Item 84: Beware of Exception Variables Disappearing

Item 85: Beware of Catching the Exception Class

Item 86: Understand the Difference Between Exception and BaseException

Item 87: Use traceback for Enhanced Exception Reporting

Item 88: Consider Explicitly Chaining Exceptions to Clarify Tracebacks

Item 89: Always Pass Resources into Generators and Have Callers Clean Them Up Outside

Item 90: Never Set __debug__ to False

Item 91: Avoid exec and eval Unless Youre Building a Developer Tool

Chapter 11 Performance

Item 92: Profile Before Optimizing

Item 93: Optimize Performance-Critical Code Using timeit Microbenchmarks

Item 94: Know When and How to Replace Python with Another Programming Language

Item 95: Consider ctypes to Rapidly Integrate with Native Libraries

Item 96: Consider Extension Modules to Maximize Performance and Ergonomics

Item 97: Rely on Precompiled Bytecode and File System Caching to Improve Startup Time

Item 98: Lazy-Load Modules with Dynamic Imports to Reduce Startup Time

Table of Contents

Item 99: Consider memoryview and bytearray for Zero-Copy Interactions with bytes

Chapter 12 Data Structures and Algorithms

Item 100: Sort by Complex Criteria Using the key Parameter

Item 101: Know the Difference Between sort and sorted

Item 102: Consider Searching Sorted Sequences with bisect

Item 103: Prefer deque for ProducerConsumer Queues

Item 104: Know How to Use heapq for Priority Queues

Item 105: Use datetime Instead of time for Local Clocks

Item 106: Use decimal when Precision Is Paramount

Item 107 : Make pickle Serialization Maintainable with copyreg

Chapter 13 Testing and Debugging

Item 108: Verify Related Behaviors in TestCase Subclasses

Item 109: Prefer Integration Tests over Unit Tests

Item 110: Isolate Tests from Each Other with setUp, tearDown, setUpModule, and tearDownModule

Item 111 : Use Mocks to Test Code with Complex Dependencies

Item 112: Encapsulate Dependencies to Facilitate Mocking and Testing

Item 113: Use assertAlmostEqual to Control Precision in Floating Point Tests

Item 114: Consider Interactive Debugging with pdb

Item 115: Use tracemalloc to Understand Memory Usage and Leaks

Chapter 14 Collaboration

Item 116: Know Where to Find Community-Built Modules

Item 117 : Use Virtual Environments for Isolated and Reproducible Dependencies

Pearson

Table of Contents

Item 118: Write Docstrings for Every Function, Class, and Module
Item 119: Use Packages to Organize Modules and Provide Stable APIs
Item 120: Consider Module-Scoped Code to Configure Deployment Environments
Item 121 : Define a Root Exception to Insulate Callers from APIs
Item 122: Know How to Break Circular Dependencies
Item 123: Consider warnings to Refactor and Migrate Usage
Item 124: Consider Static Analysis via typing to Obviate Bugs
Item 125: Prefer Open Source Projects for Bundling Python Programs over zipimport and zipapp
Index

Pearson