YZM1022

Advanced Programming

Week 6: Advanced Pythonic Programming Syntax

Instructor: Ekrem Çetinkaya

Date: 01.04.2026

The Zen of Python

Before we dive into specific idioms, let's look at Python's guiding philosophy. Run `import this` in any Python interpreter, and you get 19 aphorisms that define what *Pythonic* means:

```
import this
```

> **Beautiful is better than ugly.** - Readability counts.
> **Explicit is better than implicit.** - Don't hide what's happening.
> **Simple is better than complex.** - If you can say it in one line, do so.
> **Flat is better than nested.** - Avoid deep nesting.
> **Readability counts.** - Code is read far more often than it is written.
> **There should be one - and preferably only one - obvious way to do it.**
> **If the implementation is hard to explain, it's a bad idea.**

These are not just slogans, they are design decisions baked into the language. Every idiom we learn today is a direct application of these principles.

Today's Agenda

Data Transformation

- List, dict, and set comprehensions
- Generator expressions
- Unpacking and multiple assignment
- Slicing and advanced indexing

Modern Python Classes

- Data classes (`@dataclass`)
- Named tuples
- Slots for memory efficiency
- `__repr__` , `__eq__` , and comparison magic

Resource Management

- Context managers (`with` statement)
- Custom context managers (`__enter__` / `__exit__`)
- `contextlib` utilities

Error Handling Patterns

- Custom exception hierarchies
- EAFP vs LBYL philosophy
- Exception chaining (`from`)
- Practical error handling strategies

Comprehensions - The Pythonic Way to Transform Data

Comprehensions are one of Python's most distinctive features and they let you create new collections by transforming and filtering existing ones in a single, readable expression.

- Replace multi-line loops with concise, declarative code that says *what* you want rather than *how* to compute it step by step.

The Loop Way (Non-Pythonic)

```
# Create a list of squares
squares = []
for x in range(10):
    squares.append(x ** 2)
```

Four lines. Create empty list, loop, append. The intent is buried in the mechanics.

The Comprehension Way (Pythonic)

```
# Create a list of squares
squares = [x ** 2 for x in range(10)]
```

One line. The intent is immediately clear: *a list of x^2 for each x in 0–9*. No temporary variables, no `.append()`.

Why Comprehensions Matter

To understand why comprehensions are so important, consider a real-world data processing task.

- Suppose you have a list of student records and need to extract the names of students who passed with honors (grade ≥ 90).

Let's compare the two approaches:

```
# Without comprehension: 5 lines, imperative style
honor_students = []
for student in students:
    if student["grade"] >= 90:
        honor_students.append(student["name"])
```

```
# With comprehension: 1 line, declarative style
honor_students = [s["name"] for s in students if s["grade"] >= 90]
```

The comprehension is not just shorter also it is **easier to read** because it directly expresses the intent:

- *give me the names of students whose grade is at least 90.*
- The loop version forces you to trace through the mechanics (create list, loop, check condition, append) to understand what is being computed.

List Comprehensions - Syntax and Patterns

The general syntax is `[expression for item in iterable if condition]`.

- The `if` clause is optional and acts as a filter. You can also nest loops, though readability drops quickly beyond two levels.

```
# Basic: transform every element
names = ["alice", "bob", "charlie"]
upper_names = [name.upper() for name in names]
# ['ALICE', 'BOB', 'CHARLIE']

# With filter: only keep elements that pass a condition
even_squares = [x**2 for x in range(20) if x % 2 == 0]
# [0, 4, 16, 36, 64, 100, 144, 196, 256, 324]

# With transformation + filter
long_upper = [name.upper() for name in names if len(name) > 3]
# ['ALICE', 'CHARLIE']

# Nested loops: flatten a 2D list
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flat = [num for row in matrix for num in row]
# [1, 2, 3, 4, 5, 6, 7, 8, 9]

# Conditional expression (ternary): transform differently based on condition
labels = ["even" if x % 2 == 0 else "odd" for x in range(6)]
# ['even', 'odd', 'even', 'odd', 'even', 'odd']
```

Comprehensions - When Not to Use

If the expression or conditions become complex, a regular loop with comments is more readable. The goal is **clarity**, not cleverness.

```
result = [
    transform(item.value)
    for category in data
    for item in category.items
    if item.active and item.value > threshold
    and not item.is_deleted
    and item.owner in allowed_users
]

result = []
for category in data:
    for item in category.items:
        if not item.active or item.is_deleted:
            continue
        if item.value <= threshold:
            continue
        if item.owner not in allowed_users:
            continue
        result.append(transform(item.value))
```

Dict and Set Comprehensions

The same syntax works for dictionaries and sets.

- Dict comprehensions use `{key: value for ...}` and set comprehensions use `{value for ...}`.

```
# Dict comprehension: word -> length
words = ["hello", "world", "python", "programming"]
word_lengths = {word: len(word) for word in words}
# {'hello': 5, 'world': 5, 'python': 6, 'programming': 11}

# Dict comprehension with filter
long_words = {word: len(word) for word in words if len(word) > 5}
# {'python': 6, 'programming': 11}

# Inverting a dictionary
original = {"a": 1, "b": 2, "c": 3}
inverted = {v: k for k, v in original.items()}
# {1: 'a', 2: 'b', 3: 'c'}

# Set comprehension: unique first letters
first_letters = {word[0] for word in words}
# {'h', 'w', 'p'}

# Set comprehension with transformation
email_domains = {email.split("@")[1] for email in
                 ["a@gmail.com", "b@yahoo.com", "c@gmail.com"]}
# {'gmail.com', 'yahoo.com'}
```

Generator Expressions - Lazy Comprehensions

A generator expression looks like a list comprehension but uses parentheses instead of brackets.

- It does not create all values in memory at once, it produces them **one at a time, on demand**.
- Generators are essential for processing large datasets that would not fit in memory.

```
# List comprehension: creates entire list in memory
sum_list = sum([x**2 for x in range(1_000_000)])  # ~8 MB of memory

# Generator expression: produces values one at a time
sum_gen = sum(x**2 for x in range(1_000_000))     # ~0 bytes extra memory

# Both produce the same result, but the generator uses constant memory

# Generators work with any function that consumes iterables
max_val = max(len(line) for line in open("data.txt"))
any_negative = any(x < 0 for x in measurements)
all_valid = all(is_valid(item) for item in items)

# Chain generators for multi-step processing pipelines
lines = open("log.txt")
errors = (line for line in lines if "ERROR" in line)
timestamps = (line.split()[0] for line in errors)
recent = (ts for ts in timestamps if ts > "2026-03-01")
```

The `yield` Keyword

`yield` is the mechanism that makes generators work, and it is one of Python's most powerful features for memory-efficient data processing.

When a function contains `yield`, calling it does **not** execute the body. Instead, it returns a **generator object**.

- Each call to `next()` executes the body until the next `yield`, which produces a value and **pauses** the function, preserving its entire local state.
- The function resumes from exactly where it left off on the next `next()` call.

```
def countdown(n):
    print(f"Starting countdown from {n}")
    while n > 0:
        yield n           # Pause here, produce n
        n -= 1
        print(f"Resumed, n is now {n}")
    print("Countdown complete!")

gen = countdown(3)        # Nothing printed. Just creates generator object
print(next(gen))          # "Starting countdown from 3" -> yields 3
print(next(gen))          # "Resumed, n is now 2" -> yields 2
print(next(gen))          # "Resumed, n is now 1" -> yields 1
# next(gen)               # "Resumed, n is now 0" -> "Countdown complete!" -> StopIteration
```

Memory Comparison - List vs Generator

To make the memory difference visible, let's compare creating all numbers from 0 to 10 million:

```
import sys

# List: stores all 10 million numbers in memory at once
numbers_list = [x for x in range(10_000_000)]
print(f"List size: {sys.getsizeof(numbers_list) / 1_000_000:.1f} MB")
# List size: 89.1 MB

# Generator: stores only the recipe, not the numbers
numbers_gen = (x for x in range(10_000_000))
print(f"Generator size: {sys.getsizeof(numbers_gen)} bytes")
# Generator size: 200 bytes

# Both produce the same results when iterated
print(sum(numbers_list))  # Works - uses 89 MB
print(sum(numbers_gen))   # Works - uses ~200 bytes
```

The list uses **89 MB** while the generator uses **200 bytes**.

- This is because the generator computes each value on-the-fly and discards it after use. For data processing pipelines, file processing, and API pagination, generators are essential.

Example - Processing Large Files

```
def read_large_csv(filename):
    """Generator: yields one row at a time."""
    with open(filename) as f:
        header = f.readline().strip().split(",")
        for line in f:
            values = line.strip().split(",")
            yield dict(zip(header, values))

def filter_active(records):
    """Generator: yields only active records."""
    for record in records:
        if record.get("status") == "active":
            yield record

def extract_emails(records):
    """Generator: yields email addresses."""
    for record in records:
        yield record["email"]

records = read_large_csv("users.csv")        # Lazy
active = filter_active(records)              # Lazy
emails = extract_emails(active)              # Lazy

# Only now does processing happen, one row at a time
for email in emails:
    send_notification(email)
```

Practice - Comprehension Exercises

Questions:

1. Given `students = [("Alice", 85), ("Bob", 62), ("Charlie", 91), ("Diana", 45), ("Eve", 78)]`, write a list comprehension that returns the **names of students who passed** (grade ≥ 60).
2. Given `text = "Hello World Python Programming"`, create a **dict comprehension** mapping each word to its length: `{"Hello": 5, "World": 5, ...}`.
3. Given `matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]`, write a list comprehension that creates the **transpose**: `[[1, 4, 7], [2, 5, 8], [3, 6, 9]]`.
4. Write a **generator expression** that yields all prime numbers up to 100. Use `all(n % i != 0 for i in range(2, int(n**0.5)+1))` as the primality test.

Solution - Comprehension Exercises

```python
# 1. Names of passing students
students = [("Alice", 85), ("Bob", 62), ("Charlie", 91), ("Diana", 45), ("Eve", 78)]
passed = [name for name, grade in students if grade >= 60]
# ['Alice', 'Bob', 'Charlie', 'Eve']

# 2. Word -> length dictionary
text = "Hello World Python Programming"
word_lengths = {word: len(word) for word in text.split()}
# {'Hello': 5, 'World': 5, 'Python': 6, 'Programming': 11}

# 3. Matrix transpose
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
transpose = [[row[i] for row in matrix] for i in range(len(matrix[0]))]
# [[1, 4, 7], [2, 5, 8], [3, 6, 9]]

# 4. Prime numbers up to 100
primes = [n for n in range(2, 101)
          if all(n % i != 0 for i in range(2, int(n**0.5) + 1))]
# [2, 3, 5, 7, 11, 13, ..., 97]
```

Unpacking and Multiple Assignment

Python's unpacking syntax lets you assign multiple variables from any iterable in a single statement.

- Combined with the `*` (star) operator for collecting *the rest,* this eliminates many temporary variables and makes code more readable.

```
# Basic unpacking
x, y, z = [1, 2, 3]
first, second = "AB"

# Swap without temp variable
a, b = 1, 2
a, b = b, a  # a=2, b=1

# Star unpacking: collect the rest
first, *middle, last = [1, 2, 3, 4, 5]
# first=1, middle=[2, 3, 4], last=5

head, *tail = [1, 2, 3, 4, 5]
# head=1, tail=[2, 3, 4, 5]
```

Unpacking and Multiple Assignment

Python's unpacking syntax lets you assign multiple variables from any iterable in a single statement.

- Combined with the `*` (star) operator for collecting *the rest,* this eliminates many temporary variables and makes code more readable.

```
# Unpacking in function calls
def greet(name, age, city):
    return f"{name}, {age}, from {city}"

info = ["Alice", 30, "Istanbul"]
print(greet(*info))  # Unpacks list as positional args

config = {"name": "Bob", "age": 25, "city": "Ankara"}
print(greet(**config))  # Unpacks dict as keyword args

# Nested unpacking
(a, b), (c, d) = (1, 2), (3, 4)
```

`*args` and `**kwargs` - Flexible Function Signatures

Python functions can accept a variable number of arguments using `*args` (positional) and `**kwargs` (keyword).

- It enables you to write functions that adapt to any number of inputs, forward arguments to other functions, and build decorators and wrappers.

```
# *args: collects extra positional arguments into a tuple
def log(level, *messages):
    """Accepts any number of messages."""
    combined = " ".join(str(m) for m in messages)
    print(f"[{level}] {combined}")

log("INFO", "User", "logged", "in")          # [INFO] User logged in
log("ERROR", "Connection failed", "timeout")  # [ERROR] Connection failed timeout

# **kwargs: collects extra keyword arguments into a dict
def create_user(name, **attributes):
    """Accepts any number of named attributes."""
    user = {"name": name}
    user.update(attributes)
    return user

user = create_user("Alice", age=30, city="Istanbul", role="admin")
# {'name': 'Alice', 'age': 30, 'city': 'Istanbul', 'role': 'admin'}
```

`*args` and `**kwargs` - Forwarding and Combining

The real advantage of `*args` / `**kwargs` comes from **forwarding** arguments to other functions and combining them in decorators.

```python
# example - timing decorator
import time

def timer(func):
    """Decorator that measures execution time of any function."""
    def wrapper(*args, **kwargs):  # Accept any arguments
        start = time.time()
        result = func(*args, **kwargs)  # Forward all arguments
        elapsed = time.time() - start
        print(f"{func.__name__} took {elapsed:.3f}s")
        return result
    return wrapper

@timer
def train_model(data, epochs=10, lr=0.01):
    time.sleep(0.1)  # Simulate training
    return "trained"

train_model([1, 2, 3], epochs=5)  # train_model took 0.100s
```

`*args` and `**kwargs` - Forwarding and Combining

The real advantage of `*args` / `**kwargs` comes from forwarding arguments to other functions and **combining** them in decorators.

```
# Combining: *args/**kwargs with required parameters
def api_call(endpoint, method="GET", **params):
    """Required 'endpoint', optional 'method', any extra params."""
    query = "&".join(f"{k}={v}" for k, v in params.items())
    return f"{method} {endpoint}?{query}"

api_call("/users", method="POST", name="Alice", role="admin")
# POST /users?name=Alice&role=admin
```

Higher-Order Functions - Functions as Values

In Python, functions are **first-class objects**.

- They can be assigned to variables, passed as arguments, returned from other functions, and stored in data structures.
- A **higher-order function** is any function that takes another function as an argument or returns one.

```
# Functions are values — assign them to variables
def greet(name):
    return f"Hello, {name}!"

say_hello = greet  # No parentheses — we're assigning the function itself
print(say_hello("Alice"))  # Hello, Alice!

# Pass functions as arguments
def apply_to_each(func, items):
    """Applies func to every item and returns results."""
    return [func(item) for item in items]

names = ["alice", "bob", "charlie"]
print(apply_to_each(str.upper, names))   # ['ALICE', 'BOB', 'CHARLIE']
print(apply_to_each(len, names))         # [5, 3, 7]
```

Higher-Order Functions - Functions as Values

In Python, functions are **first-class objects**.

- They can be assigned to variables, passed as arguments, returned from other functions, and stored in data structures.
- A **higher-order function** is any function that takes another function as an argument or returns one.

```
# Return functions from functions (function factory)
def multiplier(n):
    """Returns a function that multiplies by n."""
    def multiply(x):
        return x * n
    return multiply  # Returns the inner function

double = multiplier(2)
triple = multiplier(3)
print(double(5))   # 10
print(triple(5))   # 15
```

Built-in Higher-Order Functions - map, filter, reduce

```
# map: apply a function to every element
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, numbers))       # [1, 4, 9, 16, 25]
squared = [x**2 for x in numbers]                   # Pythonic equivalent

# filter: keep elements that pass a test
evens = list(filter(lambda x: x % 2 == 0, numbers)) # [2, 4]
evens = [x for x in numbers if x % 2 == 0]          # Pythonic equivalent

# reduce: combine all elements into one value
from functools import reduce
total = reduce(lambda acc, x: acc + x, numbers)      # 15
total = sum(numbers)                                  # Pythonic equivalent

# When map/filter are better than comprehensions:
# 1. When you already have a named function
names = ["alice", "bob", "charlie"]
upper_names = list(map(str.upper, names))  # Cleaner than [n.upper() for n in names]

# 2. When processing is expensive and you want lazy evaluation
import math
results = map(math.sqrt, range(1_000_000))  # Lazy - computes on demand
first_10 = [next(results) for _ in range(10)]  # Only computes 10 values
```

Lambda Functions - Anonymous One-Liners

A **lambda** is a small anonymous function defined in a single expression. It is used when you need a quick function for a short-lived purpose (typically as an argument to `sorted`, `map`, `filter`, or as a callback).

```
# Lambda syntax: lambda arguments: expression
square = lambda x: x ** 2
add = lambda a, b: a + b

# Most common use: sorting with custom key
students = [
    {"name": "Alice", "gpa": 3.8},
    {"name": "Bob", "gpa": 3.2},
    {"name": "Charlie", "gpa": 3.9},
]
# Sort by GPA descending
by_gpa = sorted(students, key=lambda s: s["gpa"], reverse=True)

# Sort by multiple criteria: GPA descending, then name ascending
by_multi = sorted(students, key=lambda s: (-s["gpa"], s["name"]))

# Use in max/min
oldest = max(people, key=lambda p: p.age)
shortest_name = min(names, key=lambda n: len(n))
```

Slicing

Slicing is one of Python's most powerful and confusing features.

- `list[start:stop:step]` which can be combined with negative indices to handle range of data manipulation tasks without loops.

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

# Basic slicing
data[2:5]      # [2, 3, 4]
data[:3]       # [0, 1, 2] - first three
data[-3:]      # [7, 8, 9] - last three

# Step slicing
data[::2]      # [0, 2, 4, 6, 8] - every other element
data[1::2]     # [1, 3, 5, 7, 9] - every other, starting from index 1
data[::-1]     # [9, 8, 7, 6, 5, 4, 3, 2, 1, 0] - reversed

# Slice assignment (mutate in place)
data[2:5] = [20, 30, 40]  # Replace elements 2-4
data[::2] = [0]*5          # Replace every other element with 0
```

Slicing - Examples

```python
# Named slices for readability
HEADER = slice(0, 3)
FOOTER = slice(-1, None)

record = [1, 2, 3, 10, 20, 30, 40, 99]
print(record[HEADER])  # [1, 2, 3]
print(record[FOOTER])  # [99]

# Rotate a list
def rotate(lst, n):
    """Rotate list by n positions."""
    return lst[n:] + lst[:n]

print(rotate([1, 2, 3, 4, 5], 2))  # [3, 4, 5, 1, 2]

# Check if palindrome
def is_palindrome(s):
    return s == s[::-1]

print(is_palindrome("racecar"))  # True

# Chunk a list into groups of n
def chunks(lst, n):
    """Split list into chunks of size n."""
    return [lst[i:i+n] for i in range(0, len(lst), n)]

print(chunks([1, 2, 3, 4, 5, 6, 7], 3)) # [[1, 2, 3], [4, 5, 6], [7]]

# Sliding window
def windows(lst, size):
    """Generate sliding windows of given size."""
    return [lst[i:i+size] for i in range(len(lst) - size + 1)]

print(windows([1, 2, 3, 4, 5], 3)) # [[1, 2, 3], [2, 3, 4], [3, 4, 5]]
```

Data Classes - Modern Python OOP

Python 3.7 introduced `@dataclass` that automatically generates `__init__`, `__repr__`, `__eq__`, and optionally `__hash__`, `__lt__`, etc.

- This is a boilerplate for classes that are primarily a container for data (attributes).

```python
from dataclasses import dataclass, field

@dataclass
class Student:
    name: str
    age: int
    grades: list = field(default_factory=list)
    gpa: float = 0.0

    def add_grade(self, grade: float):
        self.grades.append(grade)
        self.gpa = sum(self.grades) / len(self.grades)

# Auto-generated __init__, __repr__, __eq__
alice = Student("Alice", 20)
alice.add_grade(90)
alice.add_grade(85)

print(alice)  # Student(name='Alice', age=20, grades=[90, 85], gpa=87.5)

bob = Student("Bob", 21)
print(alice == bob)  # False (compares all fields)
```

Dataclass vs Regular Class

Regular Class (30 lines)

```
class Product:
    def __init__(self, name, price, stock=0):
        self.name = name
        self.price = price
        self.stock = stock

    def __repr__(self):
        return (f"Product(name={self.name!r}, "
                f"price={self.price!r}, "
                f"stock={self.stock!r})")

    def __eq__(self, other):
        if not isinstance(other, Product):
            return NotImplemented
        return (self.name == other.name
                and self.price == other.price
                and self.stock == other.stock)

    def __hash__(self):
        return hash((self.name, self.price,
                     self.stock))
```

Dataclass (5 lines)

```
from dataclasses import dataclass

@dataclass
class Product:
    name: str
    price: float
    stock: int = 0
```

Same functionality. Five lines instead of thirty.

Dataclass is not meant to add new capabilities, but to eliminate the error-prone boilerplate that every data-holding class requires.

Data Classes - Advanced Features

Data classes support **immutability** (`frozen=True`), **ordering** (`order=True`), and **post-initialization** processing (`__post_init__`).

```python
from dataclasses import dataclass, field

@dataclass(frozen=True)
# Immutable - like a NamedTuple but with methods
class Point:
    x: float
    y: float

    def distance_to(self, other: 'Point') -> float:
        return ((self.x - other.x)**2 + (self.y - other.y)**2) ** 0.5

p1 = Point(3, 4)
# p1.x = 5
# # FrozenInstanceError! Cannot modify frozen dataclass.

@dataclass(order=True)
# Auto-generates __lt__, __le__, __gt__, __ge__
class Priority:
    priority: int
    name: str = field(compare=False)  # Excluded from comparisons

tasks = [Priority(3, "Low"), Priority(1, "Critical"), Priority(2, "Medium")]
print(sorted(tasks))
# [Priority(priority=1, name='Critical'), Priority(priority=2, name='Medium'), ...]
```

```python
@dataclass
class Config:
    host: str = "localhost"
    port: int = 8080

    def __post_init__(self):  # Runs after __init__
        if self.port < 0 or self.port > 65535:
            raise ValueError(f"Invalid port: {self.port}")
```

When to Use Which - Dataclass vs NamedTuple vs Dict vs Regular Class

With so many options for holding data in Python, it can be confusing to choose.

Need	Use
Simple data container with methods	`@dataclass`
Immutable record (like a database row)	`@dataclass(frozen=True)` or `NamedTuple`
Lightweight, tuple-compatible	`NamedTuple`
Dynamic keys (unknown at design time)	`dict`
Complex behavior with state	Regular `class`
Configuration that shouldn't change	`@dataclass(frozen=True)`
Data transfer between functions	`NamedTuple` or `@dataclass`
Need to serialize to JSON	`@dataclass` + `dataclasses.asdict()`

NamedTuple - Lightweight Immutable Records

`NamedTuple` is the older alternative to frozen dataclasses.

- It creates tuple subclasses with named fields.
- More memory-efficient than dataclasses and integrates naturally with tuple unpacking.

Good to use when you need a simple, immutable record

```
from typing import NamedTuple

class Coordinate(NamedTuple):
    latitude: float
    longitude: float
    altitude: float = 0.0

istanbul = Coordinate(41.0082, 28.9784)
ankara = Coordinate(39.9334, 32.8597, altitude=938)

# Access by name or index
print(istanbul.latitude)  # 41.0082
print(istanbul[0])        # 41.0082 - works like a tuple

# Unpacking
lat, lon, alt = istanbul

# Immutable - istanbul.latitude = 40.0  # AttributeError!
```

Practice - Data Classes

Questions:

1. Create a `@dataclass` called `Book` with fields: `title` (str), `author` (str), `pages` (int), `price` (float), `tags` (list, default empty). Add a method `is_long()` that returns `True` if pages > 300.
2. Create a `@dataclass(frozen=True)` called `RGB` with fields `r`, `g`, `b` (all int). Add a `__post_init__` that validates each value is between 0 and 255.
3. Create a `@dataclass(order=True)` called `Employee` with fields `salary` (int) and `name` (str, excluded from comparison). Create a list of 4 employees and sort them by salary.

Solution - Data Classes

```python
from dataclasses import dataclass, field
from typing import NamedTuple

# 1. Book dataclass
@dataclass
class Book:
    title: str
    author: str
    pages: int
    price: float
    tags: list = field(default_factory=list)

    def is_long(self) -> bool:
        return self.pages > 300

book = Book("PRML", "Bishop", 738, 59.99, ["ML", "Statistics"])
print(book.is_long())  # True

# 2. Frozen RGB with validation
@dataclass(frozen=True)
class RGB:
    r: int
    g: int
    b: int

    def __post_init__(self):
        for name, val in [("r", self.r), ("g", self.g), ("b", self.b)]:
            if not 0 <= val <= 255:
                raise ValueError(f"{name}={val} not in [0, 255]")
```

Solution - Data Classes

```python
# 3. Ordered Employee
@dataclass(order=True)
class Employee:
    salary: int
    name: str = field(compare=False)

employees = [Employee(5000, "Alice"), Employee(3000, "Bob"),
             Employee(7000, "Charlie"), Employee(4000, "Diana")]
print(sorted(employees))
# [Employee(salary=3000, ...), Employee(salary=4000, ...), ...]
```

Context Managers - The `with` Statement

A **context manager** is an object that defines setup and cleanup actions.

- It guarantees that cleanup happens even if an error occurs.
- The `with` statement is used to manage resources (e.g., files, database connections, locks, temporary directories, network sockets).

Without Context Manager

```
f = open("data.txt")
try:
    data = f.read()
    process(data)
finally:
    f.close()  # Must remember this
```

What if you forget `finally`? File handle leaks. What if `open()` fails? `f` is undefined and `f.close()` crashes.

With Context Manager

```
with open("data.txt") as f:
    data = f.read()
    process(data)
# File automatically closed here
```

Clean, safe, guaranteed. The file is closed whether `process()` succeeds or raises an exception.

Context Managers in Python

Context managers are used commonly in Python's standard library. Every time you use `with`, you are using a context manager.

```
# File handling - closes the file
with open("data.txt") as f:
    data = f.read()

# Thread locks - releases the lock
import threading
lock = threading.Lock()
with lock:
    shared_resource.update()

# Database transactions - commits or rolls back
with db.begin() as transaction:
    transaction.execute("INSERT INTO ...")

# Temporary directory - deletes when done
import tempfile
with tempfile.TemporaryDirectory() as tmpdir:
    save_files_to(tmpdir)

# Suppress exceptions - catches and ignores
from contextlib import suppress
with suppress(FileNotFoundError):
    os.remove("maybe_exists.txt")
```

Custom Context Managers

Any class that implements `__enter__` (setup) and `__exit__` (cleanup) can be used with `with`.

- The `__exit__` method receives exception information and runs even if an error occurs, ideal for guaranteed cleanup.

```
class DatabaseConnection:
    def __init__(self, db_name: str):
        self.db_name = db_name
        self.connection = None

    def __enter__(self):
        """Called when entering 'with' block. Returns the resource."""
        print(f"Connecting to {self.db_name}...")
        self.connection = f"Connection({self.db_name})"
        return self  # This becomes the 'as' variable

    def __exit__(self, exc_type, exc_val, exc_tb):
        """Called when leaving 'with' block. Always runs, even on error."""
        print(f"Closing connection to {self.db_name}")
        self.connection = None
        return False  # Don't suppress exceptions

    def query(self, sql: str):
        return f"Executing '{sql}' on {self.connection}"
```

Context Managers with contextlib

Python's `contextlib` module provides shortcuts for creating context managers without writing a full class.

- The `@contextmanager` decorator turns a generator function into a context manager
- Everything before `yield` is the setup, everything after is the cleanup.

```
from contextlib import contextmanager
import time

@contextmanager
def timer(label: str):
    """Context manager that measures execution time."""
    start = time.time()
    print(f"[{label}] Starting...")
    yield  # Control passes to the 'with' block here
    elapsed = time.time() - start
    print(f"[{label}] Finished in {elapsed:.3f}s")

with timer("Data processing"):
    # Simulate work
    total = sum(x**2 for x in range(1_000_000))
# [Data processing] Starting...
# [Data processing] Finished in 0.082s
```

Multiple Context Managers

Python lets you stack multiple context managers in a single `with` statement.

- This is common when you need to open an input file and an output file simultaneously, or acquire multiple locks.

```
# Multiple resources in one with statement
with open("input.txt") as fin, open("output.txt", "w") as fout:
    for line in fin:
        fout.write(line.upper())
# Both files closed here, even if an error occurs mid-processing

# Equivalent to nested with statements
with open("input.txt") as fin:
    with open("output.txt", "w") as fout:
        for line in fin:
            fout.write(line.upper())

# contextlib.ExitStack for dynamic number of context managers
from contextlib import ExitStack

filenames = ["a.txt", "b.txt", "c.txt"]
with ExitStack() as stack:
    files = [stack.enter_context(open(fn)) for fn in filenames]
    # All files open here, all closed when with block exits
    for f in files:
        print(f.readline())
```

Practice - Context Managers

Questions:

1. Write a context manager class `FileLogger` that:
   - Opens a log file in `__enter__`
   - Returns a `log(message)` method
   - Closes the file in `__exit__`
   - Usage: `with FileLogger("app.log") as logger: logger.log("Started")`
2. Write a `@contextmanager` function `suppress_errors(*exception_types)` that catches and prints specified exceptions instead of crashing:

```
with suppress_errors(ValueError, TypeError):
    int("not a number")  # Prints error but doesn't crash
```

3. Write a context manager `change_directory(path)` that temporarily changes the working directory and restores it when done, even if an error occurs.

Solution - Context Managers

`FileLogger` class-based context manager:

```
class FileLogger:
    def __init__(self, filename):
        self.filename = filename
        self.file = None

    def __enter__(self):
        self.file = open(self.filename, "a")
        return self

    def log(self, message):
        self.file.write(message + "\n")

    def __exit__(self, exc_type, exc_val, exc_tb):
        if self.file:
            self.file.close()
        return False  # Don't suppress exceptions

with FileLogger("app.log") as logger:
    logger.log("Started")
    logger.log("Processing...")
```

Solution - Context Managers

`suppress_errors` using `@contextmanager`:

```
from contextlib import contextmanager

@contextmanager
def suppress_errors(*exception_types):
    try:
        yield
    except exception_types as e:
        print(f"Suppressed {type(e).__name__}: {e}")

with suppress_errors(ValueError, TypeError):
    int("not a number")   # Suppressed ValueError: invalid literal...

with suppress_errors(ValueError, TypeError):
    None + 1              # Suppressed TypeError: ...

print("Program continues normally")
```

Solution - Context Managers

`change_directory` context manager:

```
import os
from contextlib import contextmanager

@contextmanager
def change_directory(path):
    original = os.getcwd()
    try:
        os.chdir(path)
        yield
    finally:
        os.chdir(original)  # Always restore, even on error

print(os.getcwd())          # /home/user/project

with change_directory("/tmp"):
    print(os.getcwd())      # /tmp
    # Even if an error occurs here, directory is restored

print(os.getcwd())          # /home/user/project
```

Error Handling - EAFP vs LBYL

Python has a distinctive philosophy about error handling that differs from languages like Java or C++.

- **EAFP** (Easier to Ask Forgiveness than Permission) means you *try* an operation and handle the exception if it fails, rather than checking in advance whether the operation will succeed.
- **LBYL** (Look Before You Leap) means you check *ahead of time* whether something is allowed or possible, and only proceed if it is. This is a **defensive pattern**.

> **In Python, usually EAFP is preferred**. But you can use LBYL when the check is cheap and failure is common (e.g., `if x is not None`).

Error Handling - EAFP vs LBYL

LBYL

```
if key in dictionary:
    value = dictionary[key]
else:
    value = default

if os.path.exists(filename):
    with open(filename) as f:
        data = f.read()
```

How it works: You proactively check the precondition before performing an operation. If the precondition fails, you take a different path or provide a fallback value, which avoids exceptions being raised in most cases.

Problem: There is a possible race condition because the file could be deleted or changed between the time you check and the time you operate on it.

EAFP

```
try:
    value = dictionary[key]
except KeyError:
    value = default

try:
    with open(filename) as f:
        data = f.read()
except FileNotFoundError:
    data = None
```

How it works: You optimistically assume the operation will succeed, and catch and handle any exceptions if it doesn't. This is generally more concise and avoids some forms of race conditions.

Better: Atomic operation-no race condition because the operation and failure are one step. Also, in Python, this is usually faster in the common case where no exception occurs.

The try/except/else/finally Pattern

You all probably know `try/except`, but the full pattern includes `else` (runs only if no exception) and `finally` (always runs). Understanding when to use each block makes your error handling precise and correct.

```
try:
    # Code that might fail
    result = risky_operation()
except SpecificError as e:
    # Handle the specific error
    log.error(f"Operation failed: {e}")
    result = fallback_value
except (TypeError, ValueError) as e:
    # Handle multiple exception types
    log.error(f"Bad input: {e}")
    raise  # Re-raise after logging
else:
    # Runs only if no exception occurred
    # Put success-path code here, not in try block
    save_result(result)
    notify_user("Success!")
finally:
    # Always runs - even if there's a return or raise
    cleanup_resources()
```

Custom Exceptions

Well-designed applications usually define their own exception hierarchy.

- The base exception for your application should inherit from `Exception` (never `BaseException`), and specific exceptions inherit from the base.

```python
# Base exception for the application
class AppError(Exception):
    """Base exception for our application."""
    pass

# Specific exceptions inherit from the base
class ValidationError(AppError):
    """Raised when input validation fails."""
    def __init__(self, field: str, message: str):
        self.field = field
        self.message = message
        super().__init__(f"Validation error on '{field}': {message}")

class NotFoundError(AppError):
    """Raised when a requested resource doesn't exist."""
    pass

class AuthenticationError(AppError):
    """Raised when authentication fails."""
    pass
```

Exception Chaining

Python supports **exception chaining** with `raise ... from ...`, which preserves the original error context.

- This is really useful for debugging because when you catch a low-level exception and raise a higher-level one, the original traceback is preserved.

```
# Exception chaining: preserve the original error
def get_config(filename: str) -> dict:
    try:
        with open(filename) as f:
            return json.load(f)
    except FileNotFoundError as e:
        raise ConfigError(f"Config file missing: {filename}") from e
    except json.JSONDecodeError as e:
        raise ConfigError(f"Invalid JSON in {filename}") from e

# The traceback shows both the original error and the new one:
# ConfigError: Config file missing: settings.json
#   The above exception was the direct cause of:
# FileNotFoundError: [Errno 2] No such file or directory: 'settings.json'
```

Error Handling Best Practices

These rules will save you countless hours of debugging. Memorize them.

Do:

1. **Catch specific exceptions** - `except ValueError`
2. **Use `else` for success code** - keeps try blocks minimal
3. **Fail fast** - validate inputs early, raise clear errors
4. **Log before re-raising** - `logger.error(e); raise`
5. **Chain exceptions** - `raise X from e`

Don't:

1. **Bare `except:`** - catches SystemExit, KeyboardInterrupt
2. **`except: pass`** - silences all errors including bugs
3. **Catch too broadly** - `except Exception` hides bugs
4. **Use exceptions for flow control** - if you expect it, check first
5. **Ignore the traceback** - it tells you exactly what happened

```
# The wors pattern, never do this
try:
    something()
except:       # Catches everything including Ctrl+C
    pass      # And silently ignores it
# Bugs will be invisible.
```

Practice - Error Handling

Questions:

1. Create an exception hierarchy for a banking application:
   - `BankError` (base)
   - `InsufficientFundsError(amount, balance)` - include both in the message
   - `AccountNotFoundError(account_id)`
   - `TransactionLimitError(amount, limit)`
2. Write a `BankAccount` class that uses these exceptions:
   - `withdraw(amount)` raises `InsufficientFundsError` if balance too low
   - `transfer(target, amount)` raises `TransactionLimitError` if amount > 10000
   - Use exception chaining when catching and re-raising
3. Write a function `safe_divide(a, b)` that:
   - Returns the result if successful
   - Returns `None` and prints a warning if `b` is zero
   - Uses EAFP style (try/except, not if/else)

Solution - Error Handling

Exception hierarchy for a banking application:

```
class BankError(Exception):
    """Base exception for all banking errors."""
    pass

class InsufficientFundsError(BankError):
    def __init__(self, amount, balance):
        self.amount = amount
        self.balance = balance
        super().__init__(
            f"Cannot withdraw {amount:.2f}: balance is {balance:.2f}"
        )

class AccountNotFoundError(BankError):
    def __init__(self, account_id):
        self.account_id = account_id
        super().__init__(f"Account not found: {account_id}")

class TransactionLimitError(BankError):
    def __init__(self, amount, limit):
        self.amount = amount
        self.limit = limit
        super().__init__(
            f"Transaction {amount:.2f} exceeds limit of {limit:.2f}"
        )
```

Solution - Error Handling

`BankAccount` class using the exception hierarchy:

```
class BankAccount:
    TRANSFER_LIMIT = 10_000

    def __init__(self, account_id, balance=0):
        self.account_id = account_id
        self.balance = balance

    def withdraw(self, amount):
        if amount > self.balance:
            raise InsufficientFundsError(amount, self.balance)
        self.balance -= amount

    def transfer(self, target, amount):
        if amount > self.TRANSFER_LIMIT:
            raise TransactionLimitError(amount, self.TRANSFER_LIMIT)
        try:
            self.withdraw(amount)
        except InsufficientFundsError as e:
            raise BankError(
                f"Transfer failed for account {self.account_id}"
            ) from e   # Exception chaining: original cause preserved
        target.balance += amount
```

Solution - Error Handling

`safe_divide` using EAFP style:

```
def safe_divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        print(f"Warning: division by zero (a={a})")
        return None

print(safe_divide(10, 2))   # 5.0
print(safe_divide(10, 0))   # Warning: division by zero (a=10)
                            # None
```

Python Sequences - Lists, Tuples, and Operations

A **sequence** is any ordered collection that supports indexing and iteration.

```
# Lists: mutable, most common, dynamic size
fruits = ["apple", "banana", "cherry"]
fruits.append("date")          # Add to end
fruits.insert(1, "blueberry") # Insert at index
fruits.remove("banana")        # Remove by value
popped = fruits.pop()          # Remove and return last

# Tuples: immutable, used for fixed collections
point = (3, 4)                 # Parentheses optional
rgb = 255, 128, 0              # Tuple packing
r, g, b = rgb                  # Tuple unpacking

# Strings: immutable sequence of characters
name = "Python"
print(name[0])    # 'P'
print(name[::-1]) # 'nohtyP'
# name[0] = 'J'   # TypeError! Strings are immutable

# Common sequence operations (work on ALL sequences)
len(fruits)           # Length
"apple" in fruits     # Membership test
fruits.count("apple") # Count occurrences
fruits.index("cherry") # Find index
sorted(fruits)        # New sorted list (doesn't modify original)
fruits.sort()         # Sort in place
```

List Operations

Lists are Python's workhorse data structure. Knowing these operations fluently is essential as you will use them in every program you write.

```
# Creation patterns
empty = []
zeros = [0] * 10              # [0, 0, 0, ..., 0]
range_list = list(range(5))   # [0, 1, 2, 3, 4]
copied = original.copy()      # Shallow copy (or original[:])

# Adding elements
lst.append(item)              # Add to end - O(1)
lst.extend([1, 2, 3])        # Add multiple - O(k)
lst.insert(0, item)           # Add at index - O(n) (slow for large lists!)
combined = lst1 + lst2        # Concatenate (new list)

# Removing elements
lst.remove(value)             # Remove first occurrence - O(n)
item = lst.pop()              # Remove last - O(1)
item = lst.pop(0)             # Remove first - O(n) (use deque for this!)
del lst[2]                    # Delete by index
lst.clear()                   # Remove all
```

List Operations

Lists are Python's workhorse data structure. Knowing these operations fluently is essential as you will use them in every program you write.

```
# Searching and counting
idx = lst.index(value)        # First index (raises ValueError if missing)
count = lst.count(value)      # How many times
exists = value in lst         # Membership - O(n)

# Sorting
lst.sort()                    # In-place, ascending
lst.sort(key=len, reverse=True)  # In-place, by length, descending
new_sorted = sorted(lst, key=lambda x: x.name)  # New list
```

Dictionary Operations

Dictionaries are hash-table-based, O(1) average lookup, and the backbone of Python's object system as every object's attributes are stored in a `__dict__`.

```
# Creation
empty = {}
person = {"name": "Alice", "age": 30, "city": "Istanbul"}
from_pairs = dict([("a", 1), ("b", 2)])
from_keys = dict.fromkeys(["x", "y", "z"], 0)  # All values = 0

# Access
name = person["name"]           # KeyError if missing!
name = person.get("name")       # None if missing
name = person.get("name", "Unknown")  # Default if missing

# Modification
person["email"] = "alice@example.com"  # Add or update
person.update({"age": 31, "phone": "555-1234"})  # Bulk update
person |= {"age": 32}         # Merge operator (Python 3.9+)
```

Dictionary Operations

Dictionaries are hash-table-based, O(1) average lookup, and the backbone of Python's object system as every object's attributes are stored in a `__dict__`.

```
# Deletion
del person["phone"]
age = person.pop("age")         # Remove and return
person.pop("missing", None)     # No error if missing

# Iteration
for key in person:               # Keys only
    print(key)
for key, value in person.items():  # Key-value pairs
    print(f"{key}: {value}")
for value in person.values():   # Values only
    print(value)
```

Dictionary Advanced Patterns

```
# setdefault: get value or set default (atomic operation)
groups = {}
for name, dept in [("Alice", "Eng"), ("Bob", "Sales"), ("Carol", "Eng")]:
    groups.setdefault(dept, []).append(name)
# {'Eng': ['Alice', 'Carol'], 'Sales': ['Bob']}

# Dict merging
defaults = {"theme": "dark", "lang": "en"}
user = {"lang": "tr", "font": 14}
config = defaults | user  # user overrides defaults
# {'theme': 'dark', 'lang': 'tr', 'font': 14}

# Nested dicts
data = {
    "users": {
        "alice": {"age": 30, "role": "admin"},
        "bob": {"age": 25, "role": "user"},
    }
}
# Safe nested access
role = data.get("users", {}).get("alice", {}).get("role", "unknown")

# Dictionary as switch/case
def get_handler(action):
    handlers = {
        "create": handle_create,
        "read": handle_read,
        "update": handle_update,
        "delete": handle_delete,
    }
    return handlers.get(action, handle_unknown)
```

Set Operations

Sets are unordered collections of **unique** elements. They support mathematical set operations (union, intersection, difference) that are useful for data deduplication, membership testing, and finding relationships between groups.

```
# Creation
empty = set()                # NOT {} - that's a dict!
fruits = {"apple", "banana", "cherry"}
from_list = set([1, 2, 2, 3, 3, 3])  # {1, 2, 3} - duplicates removed

# Mathematical operations
a = {1, 2, 3, 4, 5}
b = {4, 5, 6, 7, 8}

a | b   # Union: {1, 2, 3, 4, 5, 6, 7, 8}
a & b   # Intersection: {4, 5}
a - b   # Difference: {1, 2, 3}
a ^ b   # Symmetric difference: {1, 2, 3, 6, 7, 8}
```

Set Operations

Sets are unordered collections of **unique** elements. They support mathematical set operations (union, intersection, difference) that are useful for data deduplication, membership testing, and finding relationships between groups.

```
# Subset/superset testing
{1, 2} <= {1, 2, 3}   # True - subset
{1, 2, 3} >= {1, 2}   # True - superset

# Practical patterns
unique_visitors = set(log_entries)  # Deduplicate
common_friends = friends_a & friends_b  # Mutual friends
new_items = current_items - previous_items  # What changed?

# Membership testing: O(1) vs O(n) for lists!
valid_ids = {1001, 1002, 1003, 1004}  # Set lookup: O(1)
if user_id in valid_ids:              # Much faster than list
    process(user_id)
```

File I/O

File handling is one of the most common tasks in programming.

- Python makes it straightforward with the built-in `open()` function and context managers.

```
# Reading: three approaches
# 1. Read entire file into one string
with open("data.txt", "r", encoding="utf-8") as f:
    content = f.read()

# 2. Read all lines into a list
with open("data.txt") as f:
    lines = f.readlines()  # ['line1\n', 'line2\n', ...]

# 3. Iterate line by line (BEST for large files - uses constant memory)
with open("data.txt") as f:
    for line in f:
        process(line.strip())
```

File I/O

File handling is one of the most common tasks in programming.

- Python makes it straightforward with the built-in `open()` function and context managers.

```
# Writing
with open("output.txt", "w") as f:     # 'w' = overwrite
    f.write("Hello, World!\n")
    f.write("Second line\n")

with open("output.txt", "a") as f:     # 'a' = append
    f.write("Added later\n")

# Writing multiple lines
lines = ["line 1", "line 2", "line 3"]
with open("output.txt", "w") as f:
    f.writelines(line + "\n" for line in lines)
```

File I/O - JSON

Applications rarely work with plain text files. Here are the most common file formats and how to handle them in Python.

```
import json

# JSON: the universal data exchange format
data = {"name": "Ekrem", "courses": ["ML", "Advanced Programming"]}

# Write JSON
with open("data.json", "w") as f:
    json.dump(data, f, indent=2, ensure_ascii=False)  # Pretty print

# Read JSON
with open("data.json") as f:
    loaded = json.load(f)
print(loaded["courses"])  # ['ML', 'Advanced Programming']
```

File I/O - CSV

Applications rarely work with plain text files. Here are the most common file formats and how to handle them in Python.

```
import csv
# CSV: tabular data
students = [
    {"name": "Alice", "grade": 85},
    {"name": "Bob", "grade": 92},
]

# Write CSV
with open("students.csv", "w", newline="") as f:
    writer = csv.DictWriter(f, fieldnames=["name", "grade"])
    writer.writeheader()
    writer.writerows(students)

# Read CSV
with open("students.csv") as f:
    reader = csv.DictReader(f)
    for row in reader:
        print(f"{row['name']}: {row['grade']}")
```

File I/O - Path Handling with pathlib

The `pathlib` module provides an object-oriented interface for working with file paths. It is more readable and less error-prone than the old `os.path` approach.

```
from pathlib import Path

# Create path objects
home = Path.home()                    # /home/user
project = Path("my_project")
config = project / "config" / "settings.json"  # / operator joins paths!

# Check existence
config.exists()     # True/False
config.is_file()    # True/False
config.is_dir()     # True/False

# Read/write shortcuts
text = config.read_text(encoding="utf-8")
config.write_text("new content", encoding="utf-8")
```

File I/O - Path Handling with pathlib

The `pathlib` module provides an object-oriented interface for working with file paths. It is more readable and less error-prone than the old `os.path` approach.

```
# List directory contents
for path in Path(".").iterdir():
    print(f"{'DIR ' if path.is_dir() else 'FILE'} {path.name}")

# Glob: find files matching a pattern
py_files = list(Path(".").glob("**/*.py"))  # All .py files recursively
print(f"Found {len(py_files)} Python files")

# Path components
p = Path("/home/user/documents/report.pdf")
print(p.name)      # "report.pdf"
print(p.stem)      # "report"
print(p.suffix)    # ".pdf"
print(p.parent)    # "/home/user/documents"
```

Example - Data Processing Pipeline

Let's put everything together we have seen today (comprehensions, context managers, generators, and error handling) in a realistic data processing scenario.

```
from dataclasses import dataclass, field
from contextlib import contextmanager
from typing import Generator
import json

@dataclass
class ProcessingResult:
    total: int = 0
    processed: int = 0
    errors: list = field(default_factory=list)

@contextmanager
def data_pipeline(input_file: str, output_file: str):
    """Context manager that handles file I/O for a processing pipeline."""
    result = ProcessingResult()
    try:
        with open(input_file) as fin, open(output_file, 'w') as fout:
            yield fin, fout, result
    except FileNotFoundError as e:
        raise PipelineError(f"File not found: {e.filename}") from e
    finally:
        print(f"Processed {result.processed}/{result.total} "
              f"({len(result.errors)} errors)")
```

Data Pipeline - Processing Logic

```
def process_records(fin) -> Generator[dict, None, None]:
    """Generator: read, parse, and validate records one at a time."""
    for line_num, line in enumerate(fin, 1):
        try:
            record = json.loads(line.strip())
            if "email" not in record:
                raise ValidationError("email", "missing required field")
            yield record
        except json.JSONDecodeError:
            print(f"Line {line_num}: invalid JSON, skipping")

# Usage: everything together
with data_pipeline("users.jsonl", "output.jsonl") as (fin, fout, result):
    valid_records = (r for r in process_records(fin) if r.get("active"))
    enriched = (
        {**r, "domain": r["email"].split("@")[1]}
        for r in valid_records
    )
    for record in enriched:
        result.total += 1
        fout.write(json.dumps(record) + "\n")
        result.processed += 1
```

Pythonic Idioms

Below are the patterns that experienced Python developers use commonly. Each one replaces a multi-line construct with a cleaner, more readable alternative.

Iteration

```python
# Enumerate instead of range(len())
for i, item in enumerate(items):
    print(f"{i}: {item}")

# Zip for parallel iteration
for name, score in zip(names, scores):
    print(f"{name}: {score}")

# Dict iteration
for key, value in data.items():
    print(f"{key} = {value}")

# Reversed
for item in reversed(items):
    process(item)
```

Conditionals and Defaults

```python
# Ternary expression
status = "pass" if grade >= 60 else "fail"

# Default with or
name = user_input or "Anonymous"

# Default with dict.get()
value = config.get("key", "default")

# any() and all()
has_negative = any(x < 0 for x in data)
all_positive = all(x > 0 for x in data)
```

Pythonic Idioms

String Operations

```
# Join instead of += in loops
words = ["Hello", "World", "Python"]
sentence = " ".join(words)  # Not: s = ""; for w: s += w

# F-strings
name, age = "Alice", 30
msg = f"{name} is {age} years old"

# Multi-line strings
query = (
    "SELECT name, age "
    "FROM users "
    "WHERE active = true"
)
```

Collections

```
# collections.Counter
from collections import Counter
words = ["the", "cat", "sat", "on", "the", "mat"]
counts = Counter(words)
# Counter({'the': 2, 'cat': 1, ...})

# collections.defaultdict
from collections import defaultdict
groups = defaultdict(list)
for name, dept in employees:
    groups[dept].append(name)

# Sorting with key
students.sort(key=lambda s: s.gpa, reverse=True)
```

Summary

This week we focused on writing code that is not just correct, but **Pythonic** by leveraging Python's unique features to write clearer, more concise, and more maintainable code.

Data Transformation

- **Comprehensions** - list, dict, set, generator
- **Unpacking** - multiple assignment, star expressions
- **Slicing** - advanced indexing, named slices

Resource Management

- **Context managers** - `with` for guaranteed cleanup
- **@contextmanager** - generator-based shortcut
- **Custom protocols** - `__enter__` / `__exit__`

Modern Classes

- **@dataclass** - eliminate boilerplate
- **NamedTuple** - lightweight immutable records
- **frozen/ordered** dataclasses

Error Handling

- **EAFP** - try first, handle exceptions
- **Custom hierarchies** - application-specific exceptions
- **Exception chaining** - `raise ... from ...`

> The best Python code reads almost like English. If your code needs a comment to explain *what* it does (not *why*), it is probably not Pythonic enough.

Thank You!

Contact Information

- **Email:** ekrem.cetinkaya@yildiz.edu.tr
- **Office Hours:** Wednesday 13:30-15:30 - Room C-120
- **Book a slot before coming:** Booking Link
- **Course Repository:** GitHub

Next Week

Week 7: SOLID Principles and Testing