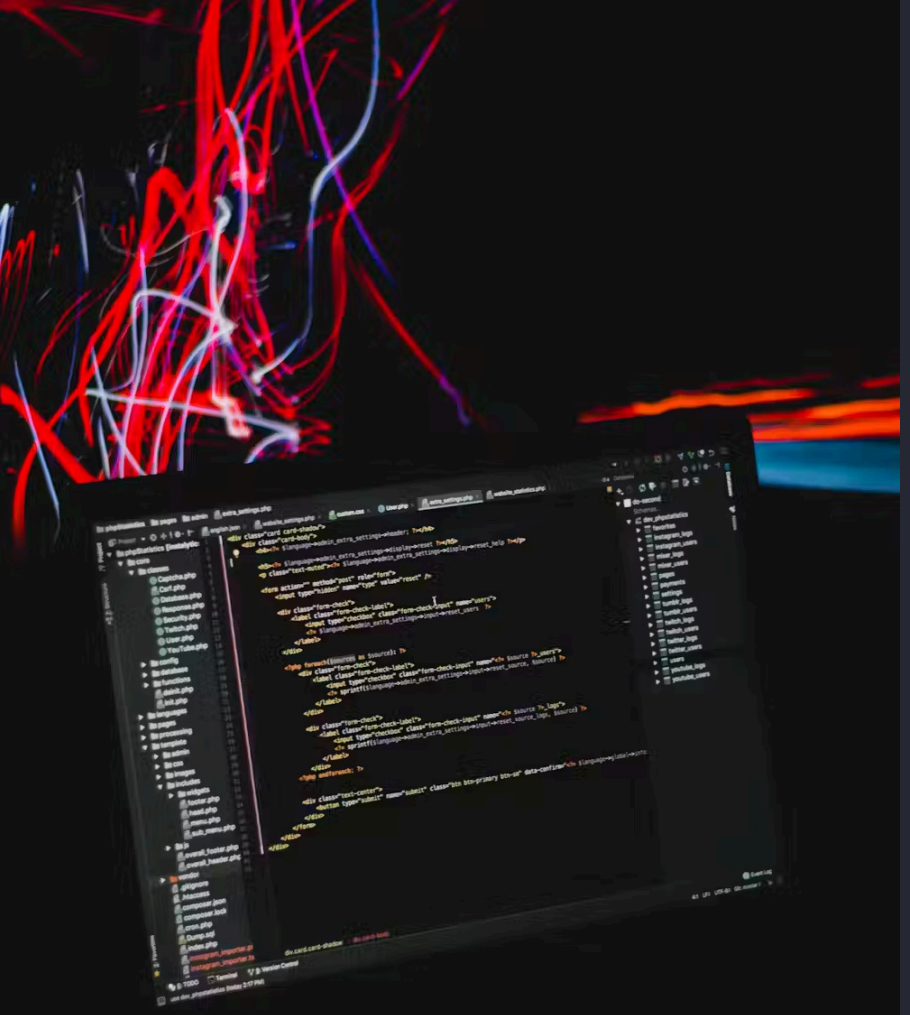


Python for Developers: Advanced OOP Features and Design Patterns

Chandrashekar Babu <training@chandrashekar.info>

https://www.chandrashekar.info/ https://www.slashprog.com/

Welcome to Day 2 of our comprehensive Python for Developers workshop. Today we'll explore advanced object-oriented programming features, design patterns, and concurrency models that will elevate your Python development skills.

Course Agenda - Day 2

1

Advanced OOP Features

Class vs instance variables, method types, inheritance, data model methods (dunder-methods), metaclasses, and abstract base classes

2

Design Patterns

Introduction to common patterns including Abstract Factory, Façade, and Chain of Responsibility

3

Context Managers

Building and implementing custom context managers for resource management

4

Iterators and Generators

Memory-efficient data structures and lazy evaluation techniques

5

Concurrency

Multi-threading, multi-processing, and asynchronous programming with coroutines

6

Exception Handling

Advanced error management techniques, custom exceptions, and assertions

Advanced OOP: Class vs Instance Attributes

Instance Attributes

- Belong to individual instances of a class
- Each instance has its own separate copy
- Defined within methods, typically in __init__
- Accessed via self.variable_name
- Used for data that varies between instances

```
class Person:
    def __init__(self, name):
        self.name = name  # Instance attribute

john = Person("John")
mary = Person("Mary")
print(john.name)  # "John"
print(mary.name)  # "Mary"
```

Class Attributes

- Belong to the class itself, shared by all instances
- Defined directly in the class body, outside any methods
- Accessed via ClassName.variable_name (preferred) or self.variable_name
- Ideal for constants or tracking class-wide data

```
class Person:
    species = "Homo Sapiens" # Class attribute
    count = 0 # Class attribute to track
              # instances

    def __init__(self, name):
        self.name = name  # Instance attribute
        Person.count += 1 # Updating class
                          # attribute

print(Person.species) # "Homo Sapiens"
print(Person.count) # Number of instances created
```

Instance Methods

Characteristics

- Most common method type in Python classes
- First parameter is always 'self' (by convention)
- 'self' refers to the instance the method is called on
- Can access and modify instance attributes
- Can access class variables but typically operate on instance data
- Called on an instance: object.method(args)

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width + self.height)

    def resize(self, width, height):
        self.width = width
        self.height = height

rect = Rectangle(5, 10)
print(rect.area())      # 50
print(rect.perimeter())  # 30
rect.resize(7, 14)
print(rect.area())      # 98
```

Instance methods can access both instance attributes (through self) and class attributes. They are the workhorses of OOP in Python, providing functionality that operates on the specific data of each instance.

Class Methods

Key Features

- Decorated with @classmethod
- First parameter is conventionally 'cls' (the class itself)
- Bound to the class, not the instance
- Can access and modify class variables
- Cannot access instance variables (no self)
- Can be called on the class or any instance

Common Use Cases

- Alternative constructors
- Factory methods
- Methods that modify class-level state
- When you need to work with the class itself

```
class Date:
    def __init__(self, day, month, year):
        self.day = day
        self.month = month
        self.year = year

    @classmethod
    def from_string(cls, date_string):
        """Alternative constructor
           from string YYYY-MM-DD"""
        year, month, day = \
           map(int, date_string.split('-'))
        return cls(day, month, year)

    @classmethod
    def today(cls):
        """Factory method for current date"""
        import datetime
        now = datetime.datetime.now()
        return cls(now.day, now.month, now.year)

# Using class methods
date1 = Date.from_string("2025-07-15")
date2 = Date.today()
```

Class methods provide a way to work with the class itself rather than instances. They're particularly useful for creating factory methods or alternative constructors that create and return instances of the class in different ways.

Static Methods

Characteristics

- Decorated with @staticmethod
- No implicit first parameter (no self or cls)
- Cannot access or modify instance or class state directly
- Behaves like a regular function that's namespaced within the class
- Can be called on the class or any instance

When to Use

- Utility functions related to the class
- Helper methods that don't need access to instance/class attributes
- Pure functions that logically belong to the class namespace
- Functions that operate on other parameters

Pythonic Considerations

- Consider creating modules with functions in them for simpler design.
- @staticmethod exists as a bridge primarily for Java / C# developers as you cannot create "functions" or "methods" outside classes in those languages.

```
class MathUtils:
    @staticmethod
    def is_prime(n):
        """Check if a number is prime"""
        for i in range(2, int(n ** 0.5)+1):
            if n % i == 0:
                return False
        return True

    @staticmethod
    def factorial(n):
        """Calculate factorial of n"""
        if n == 0 or n == 1:
            return 1
        return n * MathUtils.factorial(n-1)

# Using static methods
print(MathUtils.is_prime(17))   # True
print(MathUtils.factorial(5))   # 120
```

Static methods are essentially regular functions that are logically grouped within a class. They don't modify or access instance or class state directly, which makes them independent of the object's state. This independence makes them useful for utility functions that are conceptually related to a class but don't need to work with the class's data.

Inheritance: Building on Existing Classes

Inheritance Basics

- A mechanism where a new class (subclass) inherits attributes and methods from an existing class (superclass)
- Establishes "is-a" relationships (e.g., a Car "is a" Vehicle)
- Provides mechanisms to implement Generalization features.
- Child classes inherit all attributes and methods from parent
- Child classes can add new attributes/methods or override existing ones

Syntax

```
class BaseClass:
    # Base class attributes and methods

class DerivedClass(BaseClass):
    # Derived class attributes and methods
```

Example: Vehicle Hierarchy

```
class Vehicle:
 def __init__(self, make, model, year):
   self.make = make
   self.model = model
   self.year = year
   self.is_running = False

 def start(self):
   self.is_running = True
   print(f"{self.make} {self.model} started")

 def stop(self):
   self.is_running = False
   print(f"{self.make} {self.model} stopped")

class Car(Vehicle):
 def __init__(self, make, model,
              year, fuel_type):
   # Initialize the parent class
   super().__init__(make, model, year)
   # Add car-specific attributes
   self.fuel_type = fuel_type
   self.doors = 4

 def honk(self):
   print("Beep beep!")
```

Inheritance: Overriding and super()

Method Overriding

Subclasses can redefine methods from the parent class to provide custom behavior.

The super() Function

- Provides access to methods from the parent class
- Resolves method calls using the Method Resolution Order (MRO)
- Common in __init__ to initialize parent attributes
- Allows extending parent methods without duplication
- Syntax: super().method_name(arguments)

```
class Animal:
    def __init__(self, name, species):
        self.name = name
        self.species = species

    def make_sound(self):
        print("Some generic animal sound")

    def __str__(self):
        return f"{self.name} is a {self.species}"

class Dog(Animal):
    def __init__(self, name, breed, age):
        # Call parent's __init__
        super().__init__(name, species="Dog")
        # Add dog-specific attributes
        self.breed = breed
        self.age = age

    def make_sound(self):
        # Override parent's method
        print("Woof! Woof!")

    def __str__(self):
        # Extend parent's method
        base_str = super().__str__()
        return f"{base_str}, breed is {self.breed}, age is {self.age}"
```

Method overriding allows child classes to modify the behavior inherited from parent classes. The super() function provides a way to call methods from the parent class, which is particularly useful when you want to extend functionality rather than completely replace it.

Method Resolution Order (MRO)

What is MRO?

The order in which Python searches for methods and attributes in a class hierarchy, especially important with multiple inheritance.

MRO in Python 3

- Uses C3 Linearization algorithm
- Ensures a consistent, deterministic method resolution
- Respects the class hierarchy from left to right
- Avoids the "diamond problem" in multiple inheritance

Checking MRO

```
# View the MRO as a tuple
print(ClassName.__mro__)

# Detailed help including MRO
help(ClassName)
```

Example: Multiple Inheritance

```
class A:
    def method(self):
        print("Method from A")

class B(A):
    def method(self):
        print("Method from B")

class C(A):
    def method(self):
        print("Method from C")

class D(B, C):
    pass

# MRO for class D is:
# D -> B -> C -> A -> object
print(D.__mro__)

# When we call method() on a D instance:
d = D()
d.method()  # Prints "Method from B"
# Because B comes before C in the MRO
```

The MRO determines which method gets called when methods with the same name exist in multiple parent classes. Understanding MRO is crucial for predicting behavior in complex inheritance hierarchies.

Data Model Methods (Dunder Methods)

What are Data Model Methods?

Special methods surrounded with double underscores (__method_name__) that allow classes to implement operator support, special behaviors, and customizations.

Common Data Model Methods

- __init__(self, ...): Initializer, called when an object is created
- __str__(self): String representation for users (str(), print())
- __repr__(self): Official string representation for developers (repr())
- __len__(self): Defines behavior for len() function
- __bool__(self): Defines boolean value of an object (if obj:)
- __getattr__(self, name): Called when attribute lookup fails
- __setattr__(self, name, value): Called when setting an attribute

```
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.balance = balance

    def __str__(self):
        return f"{self.owner}'s account. Balance: £{self.balance}"

    def __repr__(self):
        return f"BankAccount('{self.owner}', {self.balance})"

    def __len__(self):
        # Could represent number of transactions
        return 1  # Placeholder

    def __bool__(self):
        # Account is "truthy" if it has a positive balance
        return self.balance > 0

    def __eq__(self, other):
        if not isinstance(other, BankAccount):
            return NotImplemented
        return self.owner == other.owner and self.balance ==
other.balance

account = BankAccount("John", 100)
print(account)      # Calls __str__
print(repr(account))  # Calls __repr__
if account:         # Calls __bool__
    print("Account has funds")
```

Data Model methods allow you to define how your objects behave with built-in functions and operators. They're a powerful way to make your classes integrate seamlessly with Python's syntax and built-in functionality.

Operator Support with Data Model Methods

Arithmetic Operators

- __add__(self, other): self + other
- __sub__(self, other): self - other
- __mul__(self, other): self * other
- __truediv__(self, other): self / other
- __floordiv__(self, other): self // other
- __mod__(self, other): self % other
- __pow__(self, other): self ** other

Comparison Operators

- __eq__(self, other): self == other
- __ne__(self, other): self != other
- __lt__(self, other): self < other
- __le__(self, other): self <= other
- __gt__(self, other): self > other
- __ge__(self, other): self >= other

Container Methods

- __getitem__(self, key): self[key]
- __setitem__(self, key, value): self[key] = value
- __delitem__(self, key): del self[key]
- __contains__(self, item): item in self
- __iter__(self): iter(self)
- __next__(self): next(self)

Other Special Methods

- __call__(self, ...): self(...) - Makes instance callable like a function
- __enter__(self), __exit__(self, exc_type, exc_val, exc_tb): Context managers
- __getattr__(self, name): Fallback for attribute access
- __getattribute__(self, name): Customized attribute access

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

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __sub__(self, other):
        return Vector(self.x - other.x, self.y - other.y)

    def __mul__(self, scalar):
        return Vector(self.x * scalar, self.y * scalar)

    def __str__(self):
        return f"Vector({self.x}, {self.y})"

v1 = Vector(1, 2)
v2 = Vector(3, 4)
v3 = v1 + v2       # Vector(4, 6)
v4 = v1 * 3        # Vector(3, 6)
```

Operator overloading makes your classes behave like built-in types, allowing for more intuitive and readable code. It's a powerful feature that lets you customize how your objects interact with Python's operators and built-in functions.

Context Managers: with Statement

What are Context Managers?

Context managers are objects that define the methods __enter__() and __exit__() to establish a context for a block of code. They're typically used with the 'with' statement.

Key Features

- Automatically set up and tear down resources
- Ensure proper cleanup even if exceptions occur
- Simplify code by removing try/finally boilerplate
- Make resource management more predictable

Common Use Cases

- File operations (opening and closing files)
- Database connections
- Network connections
- Locks in threaded programs
- Temporary context changes

The with Statement

```
# Traditional approach
file = open('example.txt', 'w')
try:
    file.write('Hello, world!')
finally:
    file.close()

# With context manager
with open('example.txt', 'w') as file:
    file.write('Hello, world!')
# File is automatically closed when exiting the block
```

How Context Managers Work

1. The with statement calls the context manager's __enter__ method
2. The value returned by __enter__ is assigned to the variable after 'as'
3. The code block inside the with statement executes
4. When the block exits (normally or by exception), __exit__ is called
5. If an exception occurred, it's passed to __exit__
6. If __exit__ returns True, the exception is suppressed

Implementing Context Managers

Class-based Context Manager

Implement __enter__ and __exit__ methods in a class.

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

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

    def __exit__(self, exc_type, exc_val, exc_tb):
        if self.file:
            self.file.close()
        # Return False to let exceptions propagate
        # Return True to suppress exceptions
        return False

# Usage
with FileManager('example.txt', 'w') as file:
    file.write('Hello, world!')
```

Function-based Context Manager with contextlib

Use the @contextmanager decorator with a generator function.

```
from contextlib import contextmanager

@contextmanager
def file_manager(filename, mode):
    try:
        # Code before yield is like __enter__
        file = open(filename, mode)
        yield file  # This value is returned by with ... as
    finally:
        # Code after yield is like __exit__
        file.close()

# Usage
with file_manager('example.txt', 'w') as file:
    file.write('Hello, world!')
```

Other contextlib Utilities

- suppress: Suppress specific exceptions
- closing: Close object on context exit
- nullcontext: A context manager that does nothing
- ExitStack: Dynamically manage multiple context managers

```
# A practical example: Database connection context manager
class DatabaseConnection:
 def __init__(self, connection_string):
    self.connection_string = connection_string
    self.connection = None

 def __enter__(self):
    # Code for connecting to the database
    print(f"Connecting to database: {self.connection_string}")
    self.connection = {"connected": True} # Mock connection
    return self

 def __exit__(self, exc_type, exc_val, exc_tb):
    # Code for closing the connection
    if self.connection:
        print("Closing database connection")
        self.connection = None

    if exc_type is not None:
        print(f"An error occurred: {exc_val}")
        # Log the error, perform cleanup, etc.

        # Return False to propagate exceptions
        return False

# Usage
with DatabaseConnection("postgresql://user:password@localhost/db") as conn:
    # Database operations
    print("Performing database operations")
    # If an exception occurs here, __exit__ will be called with the exception info
```

Iterators and Iterables

Iterables

An object capable of returning its elements one at a time.

Key Characteristics

- Implements the __iter__ method
- __iter__ returns an iterator
- Examples: lists, tuples, dictionaries, sets, strings
- Can be used in for loops, comprehensions, and functions like map(), filter()

Iterators

An object representing a stream of data.

Key Characteristics

- Implements __iter__ (returns self) and __next__ methods
- __next__ returns the next item or raises StopIteration
- Maintains state between calls to __next__
- Once exhausted, it remains exhausted

Basic Iterator Implementation

```
class CountDown:
    def __init__(self, start):
        self.start = start

    def __iter__(self):
        # Return an iterator (CountDownIterator)
        return CountDownIterator(self.start)

class CountDownIterator:
    def __init__(self, count):
        self.count = count

    def __iter__(self):
        # Iterator must also be iterable,
        # and return self
        return self

    def __next__(self):
        if self.count <= 0:
            raise StopIteration
        self.count -= 1
        return self.count + 1

# Usage
for i in CountDown(5):
    print(i) # 5, 4, 3, 2, 1

# We can also get the iterator manually
iterator = iter(CountDown(3))
print(next(iterator)) # 3
print(next(iterator)) # 2
print(next(iterator)) # 1
# next(iterator) # Raises StopIteration
```

Generators: Memory-Efficient Iterators

What are Generators?

Generators are a special type of iterator created using functions with the yield statement. They allow you to create iterators with minimal code.

Key Features

- Use yield statement to produce a sequence of values
- State is automatically saved between yields
- Memory efficient - values are generated on demand
- Can represent infinite sequences
- Simpler than writing iterator classes

Generator Function Example

```
def countdown(start):
    while start > 0:
        yield start
        start -= 1

# Usage
for i in countdown(5):
    print(i)  # 5, 4, 3, 2, 1

# We can also get the generator manually
gen = countdown(3)
print(next(gen))  # 3
print(next(gen))  # 2
print(next(gen))  # 1
# next(gen)  # Raises StopIteration
```

Advanced Generator Features

yield from

Delegates to another generator or iterable.

```
def combined_generators():
    yield from range(3)  # 0, 1, 2
    yield from "abc"     # 'a', 'b', 'c'
    yield from [7, 8, 9]  # 7, 8, 9

list(combined_generators())  # [0, 1, 2, 'a', 'b', 'c', 7, 8, 9]
```

Sending Values to Generators

```
def echo_generator():
    response = yield
    while True:
        response = yield f"Echo: {response}"

gen = echo_generator()
next(gen)  # Prime the generator
print(gen.send("Hello"))  # "Echo: Hello"
print(gen.send("World"))  # "Echo: World"
```

Generator Methods

- send(value): Send a value into the generator
- throw(type, value, traceback): Raise an exception inside the generator
- close(): Close the generator (raises GeneratorExit)

Practical Example: File Reader Generator

```
def read_large_file(file_path, chunk_size=1024):
    """Generator to read a large file in chunks."""
    with open(file_path, 'r') as file:
        while True:
            # Read a chunk of data
            data = file.read(chunk_size)

            # If no more data, stop iteration
            if not data:
                break

            # Yield the chunk
            yield data

# Usage for processing a large file without loading it all into memory
for chunk in read_large_file('large_log_file.txt'):
    process_data(chunk)
```

Generators provide a powerful way to work with data streams and large datasets in a memory-efficient manner. They're especially useful when processing large files, working with infinite sequences, or implementing data pipelines.

Comprehensions: Concise Data Construction

List Comprehension

A concise way to create lists based on existing iterables.

```
# Syntax
[expression for item in iterable if condition]

# Example: Squares of even numbers from 0-9
squares = [x**2 for x in range(10) if x % 2 == 0]
print(squares)  # [0, 4, 16, 36, 64]

# Equivalent for loop
squares = []
for x in range(10):
    if x % 2 == 0:
        squares.append(x**2)
```

Dictionary Comprehension

Create dictionaries using a similar syntax.

```
# Syntax
{key_expr: value_expr for item in iterable if condition}

# Example: Map numbers to their squares
square_dict = {x: x**2 for x in range(5)}
print(square_dict)  # {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

# Filtering example
even_square_dict = {x: x**2 for x in range(10) if x % 2 == 0}
```

Set Comprehension

Create sets using comprehension syntax.

```
# Syntax
{expression for item in iterable if condition}

# Example: Set of unique letters in a string
letters = {char.lower() for char in "Hello World" if
char.isalpha()}
print(letters)  # {'e', 'd', 'h', 'l', 'o', 'r', 'w'}
```

Nested Comprehensions

Create more complex data structures with nested loops.

```
# Flattening a 2D list
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flattened = [x for row in matrix for x in row]
print(flattened)  # [1, 2, 3, 4, 5, 6, 7, 8, 9]

# Creating a matrix
matrix = [[i * j for j in range(1, 4)] for i in range(1, 4)]
print(matrix)  # [[1, 2, 3], [2, 4, 6], [3, 6, 9]]
```

Benefits and Best Practices

- **Readability**: More concise and often clearer than equivalent for loops
- **Performance**: Generally faster than manual list building with for loops
- **Expressiveness**: Focuses on what you want, not how to get it
- **Caution**: Avoid complex or overly nested comprehensions that reduce readability
- **Memory usage**: Comprehensions build the entire structure at once, which might be inefficient for very large data sets

Generator Expressions: Lazy Comprehensions

What are Generator Expressions?

Generator expressions are similar to list comprehensions but create generators instead of lists. They use parentheses instead of square brackets.

Key Features

- Lazy evaluation - elements are produced on demand
- Memory efficient - doesn't store the entire sequence
- Can be used with very large or infinite sequences
- Ideal for pipeline processing of data
- Can only be iterated over once

Syntax Comparison

```
# List comprehension (eager, stores all values)
[x**2 for x in range(1000000)]

# Generator expression (lazy, computes values on demand)
(x**2 for x in range(1000000))
```

Memory Efficiency Example

```
import sys

# List comprehension
squares_list = [x**2 for x in range(10000)]
print(f"List size: {sys.getsizeof(squares_list)} bytes")

# Generator expression
squares_gen = (x**2 for x in range(10000))
print(f"Generator size: {sys.getsizeof(squares_gen)} bytes")

# The generator is much smaller in memory!
```

Use in Function Arguments

Generator expressions can be used directly as function arguments.

```
# Sum of all squares from 0-999
result = sum(x**2 for x in range(1000))

# Find maximum value
max_value = max(len(word) for word in words)

# Check if any items match a condition
has_even = any(x % 2 == 0 for x in numbers)
```

Pipeline Processing with Generator Expressions

```
def process_log_file(filename):
    # Open the file and read lines
    with open(filename, 'r') as file:
        # Get only lines with ERROR
        error_lines = (line for line in file if "ERROR" in line)

        # Extract timestamps from error lines
        timestamps = (line.split()[0] for line in error_lines)

        # Parse timestamps to datetime objects
        from datetime import datetime
        dates = (datetime.strptime(ts, "%Y-%m-%d") for ts in timestamps)

        # Count errors by month
        month_counts = {}
        for date in dates:
            month_key = (date.year, date.month)
            month_counts[month_key] = month_counts.get(month_key, 0) + 1

        return month_counts

# This processes a potentially huge log file using minimal memory
```

Generator expressions are ideal for data processing pipelines and working with large datasets. They allow you to express complex data transformations concisely while maintaining memory efficiency through lazy evaluation.

Exception Handling: Graceful Error Management

What are Exceptions?

Exceptions are events that occur during program execution that disrupt the normal flow of instructions. They represent errors or exceptional conditions that need special handling.

Why Handle Exceptions?

- Prevent abrupt program termination
- Provide meaningful error messages
- Implement recovery strategies
- Ensure proper resource cleanup
- Separate error handling from main logic

Common Built-in Exceptions

- TypeError: Inappropriate argument type
- ValueError: Appropriate type but inappropriate value
- FileNotFoundError: File or directory doesn't exist
- ZeroDivisionError: Division by zero
- IndexError: Index out of range
- KeyError: Key not found in dictionary
- AttributeError: Attribute not found

Exception Hierarchy

```
BaseException
 ├── SystemExit
 ├── KeyboardInterrupt
 ├── GeneratorExit
 └── Exception
      ├── StopIteration
      ├── ArithmeticError
      │    ├── FloatingPointError
      │    ├── OverflowError
      │    └── ZeroDivisionError
      ├── AssertionError
      ├── AttributeError
      ├── EOFError
      ├── ImportError
      │    └── ModuleNotFoundError
      ├── LookupError
      │    ├── IndexError
      │    └── KeyError
      ├── NameError
      ├── OSError
      │    ├── FileNotFoundError
      │    ├── PermissionError
      │    └── ...
      ├── TypeError
      ├── ValueError
      └── ... (many more)
```

try, except, else, finally Blocks

Basic Structure

```
try:
    # Code that might raise an exception
    result = x / y
except ZeroDivisionError as e:
    # Handle specific exception
    print(f"Error: {e}")
except (TypeError, ValueError) as e:
    # Handle multiple exceptions
    print(f"Input error: {e}")
except Exception as e:
    # Catch-all for other exceptions (use sparingly)
    print(f"Unexpected error: {e}")
else:
    # Execute if no exceptions were raised
    print(f"Result: {result}")
finally:
    # Always execute, regardless of exceptions
    print("Cleanup code")
```

try Block

Contains code that might raise exceptions.

except Block

Handles specific exceptions. Can have multiple except blocks for different exceptions. The first matching except block is executed.

else Block

Executes only if no exceptions were raised in the try block. Useful for code that should run only if the try block succeeds.

finally Block

Always executes, regardless of whether an exception occurred. Used for cleanup operations like closing files or releasing resources.

Exception Propagation

If an exception is not caught, it propagates up the call stack until caught or terminates the program.

```
def function_a():
    try:
        function_b()
    except ValueError:
        print("Caught ValueError in function_a")

def function_b():
    function_c()  # Will propagate exceptions

def function_c():
    raise ValueError("Error in function_c")

function_a()  # Prints "Caught ValueError in function_a"
```

Practical Example: File Processing with Exception Handling

```
def process_file(filename):
    try:
        with open(filename, 'r') as file:
            data = file.read()
            result = len(data.split())
            return result
    except FileNotFoundError:
        print(f"Error: File '{filename}' not found")
        # Could create the file or use a default
        return 0
    except PermissionError:
        print(f"Error: No permission to read '{filename}'")
        # Could request elevated permissions
        return 0
    except Exception as e:
        print(f"Unexpected error: {e}")
        # Log the error for debugging
        raise  # Re-raise the exception for higher-level handling
    else:
        print(f"Successfully processed file: {filename}")
    finally:
        print("File processing attempt completed")

# Usage
word_count = process_file("sample.txt")
```

Effective exception handling makes your code more robust and user-friendly. It allows you to anticipate and handle errors gracefully, providing appropriate feedback and recovery mechanisms.

Creating Custom Exceptions

Why Create Custom Exceptions?

- Provide more specific error types for your application
- Make error handling more precise and meaningful
- Improve code readability and maintainability
- Allow clients to catch only your application's errors
- Include application-specific data in exceptions

Basic Custom Exception

```
class MyCustomError(Exception):
    """Base exception for my application."""
    pass

class ValueTooLargeError(MyCustomError):
    """Raised when the input value is too large."""
    pass

class ValueTooSmallError(MyCustomError):
    """Raised when the input value is too small."""
    pass

# Usage
def check_value(value):
    if value > 100:
        raise ValueTooLargeError("Value must be <= 100")
    if value < 0:
        raise ValueTooSmallError("Value must be >= 0")
    return value
```

Adding Context to Custom Exceptions

```
class ValidationError(Exception):
    """Exception raised for validation errors."""

    def __init__(self, message, field=None, value=None):
        self.message = message
        self.field = field
        self.value = value
        super().__init__(self.message)

    def __str__(self):
        error_msg = self.message
        if self.field:
            error_msg += f" (field: {self.field}"
            if self.value is not None:
                error_msg += f", value: {self.value}"
            error_msg += ")"
        return error_msg

# Usage
def validate_user(user_data):
    if 'email' not in user_data:
        raise ValidationError("Email is required", "email")

    email = user_data['email']
    if '@' not in email:
        raise ValidationError(
            "Invalid email format", "email", email
        )
```

Building an Exception Hierarchy

```
# Base exception for the application
class AppError(Exception):
    """Base class for all exceptions in this application."""
    pass

# Database exceptions
class DatabaseError(AppError):
    """Base class for database-related exceptions."""
    pass

class ConnectionError(DatabaseError):
    """Raised when database connection fails."""
    pass

class QueryError(DatabaseError):
    """Raised when a database query fails."""
    def __init__(self, message, query=None, params=None):
        self.query = query
        self.params = params
        super().__init__(message)

# Authentication exceptions
class AuthError(AppError):
    """Base class for authentication-related exceptions."""
    pass

class LoginError(AuthError):
    """Raised when login fails."""
    pass

class PermissionError(AuthError):
    """Raised when user doesn't have permission."""
    pass

# Usage with custom exception handling
try:
    # Code that might raise our custom exceptions
    result = execute_query("SELECT * FROM users WHERE id = %s", [user_id])
except ConnectionError as e:
    # Handle connection issues
    reconnect_to_database()
except QueryError as e:
    # Log the problematic query
    logger.error(f"Query failed: {e.query} with params {e.params}")
except DatabaseError as e:
    # Handle other database issues
    pass
except AppError as e:
    # Catch all other application errors
    pass
```

Custom exceptions make your code more expressive and your error handling more precise. They help communicate the nature of errors more clearly and allow for more specific handling strategies.

Asserts: Debugging and Preconditions

What are Assertions?

Assertions are statements that check if a condition is true. If false, they raise an AssertionError. They're primarily used for debugging and verifying assumptions during development.

Basic Syntax

```
assert condition[, error_message]

# Examples
def calculate_average(numbers):
    assert len(numbers) > 0, "Cannot calculate average of empty list"
    return sum(numbers) / len(numbers)

def set_age(person, age):
    assert isinstance(age, int), "Age must be an integer"
    assert 0 <= age <= 150, "Age must be between 0 and 150"
    person['age'] = age
```

When to Use Assertions

- Checking preconditions at the start of functions
- Verifying invariants in your code
- Validating internal assumptions
- Post-condition checking at function exit
- Catching programming errors during development

Assertions vs. Exceptions

Assertions

- For debugging and development
- Check programming errors and assumptions
- Can be disabled in production (python -O)
- Should never be used for validating user input
- Not meant to handle expected runtime conditions

Exceptions

- For runtime error handling
- Handle expected error conditions
- Always active, even in production
- Appropriate for validating user input
- Should include recovery strategies

Python -O Flag

When Python is run with the -O flag (optimized mode), all assert statements are ignored. This means you should never use assertions for code that must run in production.

```
# Command line: python -O script.py
# All assertions will be skipped!
```

Practical Assertion Examples

```
def binary_search(sorted_list, item):
    """Find item in a sorted list using binary search."""
    # Precondition: list must be sorted
    assert all(sorted_list[i] <= sorted_list[i+1] for i in range(len(sorted_list)-1)), \
        "List must be sorted for binary search"

    low = 0
    high = len(sorted_list) - 1

    while low <= high:
        mid = (low + high) // 2
        guess = sorted_list[mid]

        if guess == item:
            return mid
        if guess > item:
            high = mid - 1
        else:
            low = mid + 1

    return None
```

```
def divide_safely(a, b):
    """Divide a by b, with assertions for type checking."""
    # Type checking
    assert isinstance(a, (int, float)), "Dividend must be a number"
    assert isinstance(b, (int, float)), "Divisor must be a number"

    # Prevent division by zero
    assert b != 0, "Cannot divide by zero"

    result = a / b

    # Post-condition: result should be a number
    assert isinstance(result, (int, float)), "Result is not a number"

    return result
```

Assertions are valuable tools for catching programming errors early in the development process. They help document and enforce assumptions in your code, making it more robust and easier to debug.

Descriptors and Properties

Properties

Properties provide a way to customize access to instance attributes. They're implemented as special methods but accessed like attributes.

```
class Person:
    def __init__(self, name, age):
        self._name = name
        self._age = age

    @property
    def name(self):
        """Getter method for name."""
        return self._name

    @name.setter
    def name(self, value):
        """Setter method for name."""
        if not isinstance(value, str):
            raise TypeError("Name must be a string")
        if len(value) < 2:
            raise ValueError("Name must be at least 2 characters")
        self._name = value

    @property
    def age(self):
        """Getter method for age."""
        return self._age

    @age.setter
    def age(self, value):
        """Setter method for age."""
        if not isinstance(value, int):
            raise TypeError("Age must be an integer")
        if value < 0 or value > 150:
            raise ValueError("Age must be between 0 and 150")
        self._age = value

    @property
    def is_adult(self):
        """Read-only property."""
        return self._age >= 18

# Usage
person = Person("John", 30)
print(person.name)      # Calls name getter
person.name = "Jane"    # Calls name setter
print(person.is_adult)  # True (read-only property)
```

Descriptors

Descriptors are objects that define how attribute access is handled. They implement __get__, __set__, and/or __delete__ methods.

```
class Validator:
    """A descriptor for validating attributes."""

    def __init__(self, type_=None, min_=None, max_=None):
        self.type = type_
        self.min = min_
        self.max = max_
        self.name = None  # Set by __set_name__

    def __set_name__(self, owner, name):
        """Set the name of the attribute."""
        self.name = name

    def __get__(self, instance, owner):
        """Return the attribute value."""
        if instance is None:
            return self
        return instance.__dict__[self.name]

    def __set__(self, instance, value):
        """Validate and set the attribute value."""
        if self.type is not None and not isinstance(value,
self.type):
            raise TypeError(f"{self.name} must be of type
{self.type.__name__}")

        if self.min is not None and value < self.min:
            raise ValueError(f"{self.name} must be >= {self.min}")

        if self.max is not None and value > self.max:
            raise ValueError(f"{self.name} must be <= {self.max}")

        instance.__dict__[self.name] = value

class Person:
    name = Validator(type_=str)
    age = Validator(type_=int, min_=0, max_=150)

    def __init__(self, name, age):
        self.name = name
        self.age = age

# Usage
person = Person("John", 30)
person.age = 35  # Valid
# person.age = -5  # ValueError: age must be >= 0
# person.age = "35"  # TypeError: age must be of type int
```

Data Descriptors vs. Non-Data Descriptors

- **Data Descriptors**: Implement both __get__ and __set__ methods. Take precedence over instance dictionary.
- **Non-Data Descriptors**: Implement only __get__ method. Instance dictionary takes precedence.

Use Cases for Descriptors

- **Validation**: Ensure attributes meet specific criteria
- **Type Conversion**: Automatically convert values to the right type
- **Lazy Loading**: Compute or fetch values only when accessed
- **Logging**: Track when attributes are accessed or modified
- **Computed Properties**: Calculate values on-the-fly

```
class LazyAttribute:
"""A descriptor that computes a value only when first accessed."""

def __init__(self, function):
self.function = function
self.name = function.__name__

def __get__(self, instance, owner):
if instance is None:
return self

# Compute the value and store it in the instance dict
value = self.function(instance)
instance.__dict__[self.name] = value

return value

class DataProcessor:
def __init__(self, data):
self.data = data

@LazyAttribute
def processed_data(self):
print("Processing data (expensive operation)...")
import time
time.sleep(2) # Simulate expensive computation
return [x * 2 for x in self.data]

@LazyAttribute
def data_stats(self):
print("Calculating statistics...")
return {
'min': min(self.data),
'max': max(self.data),
'avg': sum(self.data) / len(self.data)
}

# Usage
processor = DataProcessor([1, 2, 3, 4, 5])
# No processing happens yet

print("Accessing processed data first time:")
print(processor.processed_data) # Computation happens now

print("Accessing processed data second time:")
print(processor.processed_data) # No computation, returns cached value
```

Functional Programming in Python

First-Class Functions

In Python, functions are first-class objects, meaning they can be passed around like any other object.

```
# Assigning functions to variables
def greet(name):
    return f"Hello, {name}!"

say_hello = greet
print(say_hello("John"))  # "Hello, John!"

# Functions as arguments
def apply_function(func, value):
    return func(value)

def square(x):
    return x * x

def double(x):
    return x * 2

print(apply_function(square, 5))  # 25
print(apply_function(double, 5))  # 10

# Functions returning functions
def create_multiplier(factor):
    def multiplier(x):
        return x * factor
    return multiplier

double = create_multiplier(2)
triple = create_multiplier(3)

print(double(5))  # 10
print(triple(5))  # 15
```

Higher-Order Functions

Functions that take other functions as arguments or return functions.

Built-in Higher-Order Functions

```
# map: Apply function to each item in an iterable
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, numbers))
print(squared)  # [1, 4, 9, 16, 25]

# filter: Keep items that match a condition
evens = list(filter(lambda x: x % 2 == 0, numbers))
print(evens)  # [2, 4]

# reduce: Reduce iterable to single value by applying function
from functools import reduce
product = reduce(lambda x, y: x * y, numbers)
print(product)  # 120 (1*2*3*4*5)

# sorted: Sort with custom key function
names = ["Alice", "Bob", "Charlie", "Dave"]
sorted_by_length = sorted(names, key=len)
print(sorted_by_length)  # ["Bob", "Dave", "Alice", "Charlie"]
```

Lambda Functions

Small anonymous functions created with the lambda keyword.

```
# Basic lambda function
add = lambda x, y: x + y
print(add(5, 3))  # 8

# Lambda with filter
data = [("Alice", 25), ("Bob", 30), ("Charlie", 35), ("Dave", 40)]
young_people = list(filter(lambda person: person[1] < 35, data))
print(young_people)  # [("Alice", 25), ("Bob", 30)]

# Lambda with map
names_only = list(map(lambda person: person[0], data))
print(names_only)  # ["Alice", "Bob", "Charlie", "Dave"]

# Lambda with sorting
sorted_by_age = sorted(data, key=lambda person: person[1])
print(sorted_by_age)  # [("Alice", 25), ("Bob", 30), ("Charlie", 35), ("Dave", 40)]
```

Functional Concepts in Python

- **Pure Functions**: Functions with no side effects that return the same output for the same input
- **Immutability**: Using immutable data structures (tuples, frozensets) to avoid side effects
- **Function Composition**: Combining functions to create new functions
- **Recursion**: Functions that call themselves to solve problems

```
# Function composition
def compose(f, g):
 return lambda x: f(g(x))

add_five = lambda x: x + 5
multiply_by_three = lambda x: x * 3

add_then_multiply = compose(multiply_by_three, add_five)
multiply_then_add = compose(add_five, multiply_by_three)

print(add_then_multiply(10)) # (10 + 5) * 3 = 45
print(multiply_then_add(10)) # (10 * 3) + 5 = 35
```

Decorators: Advanced Usage

Basic Decorator Structure

Decorators are functions that modify the behavior of other functions.

```
def simple_decorator(func):
    def wrapper(*args, **kwargs):
        print("Before function call")
        result = func(*args, **kwargs)
        print("After function call")
        return result
    return wrapper

@simple_decorator
def greet(name):
    print(f"Hello, {name}!")

# Equivalent to:
# greet = simple_decorator(greet)

greet("John")
# Output:
# Before function call
# Hello, John!
# After function call
```

Decorators with Arguments

```
def repeat(n=1):
    def decorator(func):
        def wrapper(*args, **kwargs):
            result = None
            for _ in range(n):
                result = func(*args, **kwargs)
            return result
        return wrapper
    return decorator

@repeat(3)
def say_hello():
    print("Hello!")

say_hello()
# Output:
# Hello!
# Hello!
# Hello!
```

Class-based Decorators

```
class Timer:
    def __init__(self, func):
        self.func = func
        self.calls = 0
        self.total_time = 0

    def __call__(self, *args, **kwargs):
        import time
        self.calls += 1
        start = time.time()
        result = self.func(*args, **kwargs)
        end = time.time()
        self.total_time += (end - start)
        print(f"Call {self.calls}: {end - start:.4f} seconds")
        return result

    def avg_time(self):
        return self.total_time / self.calls if self.calls else 0

@Timer
def slow_function(n):
    import time
    time.sleep(n)
    return n * n

slow_function(0.5)  # Call 1: 0.5000 seconds
slow_function(1.0)  # Call 2: 1.0000 seconds
print(f"Average time: {slow_function.avg_time():.4f} seconds")
```

Practical Decorator Examples

```
# Memoization decorator for caching results
def memoize(func):
    cache = {}

    def wrapper(*args):
        if args not in cache:
            cache[args] = func(*args)
        return cache[args]

    wrapper.cache = cache  # Access cache externally if needed
    return wrapper

@memoize
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

# Without memoization, this would be extremely slow
print(fibonacci(100))  # Fast calculation due to caching

# Rate limiting decorator
import time
from functools import wraps

def rate_limit(max_calls, period):
    """Limit function to max_calls per period seconds."""
    calls = []

    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            now = time.time()

            # Remove calls older than the period
            while calls and calls[0] < now - period:
                calls.pop(0)

            # Check if we've reached the limit
            if len(calls) >= max_calls:
                raise Exception(f"Rate limit exceeded: {max_calls} calls per {period} seconds")

            # Add current call time
            calls.append(now)

            # Call the function
            return func(*args, **kwargs)

        return wrapper

    return decorator

@rate_limit(max_calls=2, period=5)
def api_request(endpoint):
    print(f"Requesting {endpoint}...")
    # Actual API call would go here
    return {"status": "success"}

# This would work
api_request("/users")
api_request("/products")

# This would raise an exception (rate limit exceeded)
# api_request("/orders")
```

Stacking Decorators

```
@decorator1
@decorator2
@decorator3
def function():
 pass

# Equivalent to:
# function = decorator1(decorator2(decorator3(function)))
```

Metaclasses: Classes of Classes

What are Metaclasses?

Metaclasses are classes whose instances are classes themselves. They define how classes are created, just as classes define how objects are created.

Key Concepts

- In Python, classes are objects too
- Every class is an instance of a metaclass
- The default metaclass is 'type'
- type(object) gives the class of an object
- type(class) gives the metaclass of a class

Use Cases

- Enforcing coding standards or patterns
- Automatic registration of classes
- Adding/modifying methods and attributes at class creation time
- Creating domain-specific languages (DSLs)
- Framework and API design

Creating Classes with type()

```
# Normal class definition
class Dog:
    def bark(self):
        return "Woof!"

# Equivalent using type()
Dog = type('Dog', (), {
    'bark': lambda self: "Woof!"
})

# Creating an instance
dog = Dog()
print(dog.bark())  # "Woof!"
```

Basic Metaclass Example

```
class Meta(type):
 def __new__(mcs, name, bases, attrs):
 # Add a class attribute to all classes
 attrs['added_by_meta'] = True

 # Log when a class is created
 print(f"Creating class: {name}")

 # Return the new class
 return super().__new__(mcs, name, bases, attrs)

# Use the metaclass
class MyClass(metaclass=Meta):
 pass

print(MyClass.added_by_meta) # True
```

Creating a Custom Metaclass

Metaclass Components

- __new__: Called to create a new class object
- __init__: Called to initialize the newly created class
- __call__: Called when the class is called to create an instance

Applying a Metaclass

```
# Method 1: Using metaclass keyword
class MyClass(metaclass=MyMeta):
    pass

# Method 2: For Python 2 compatibility
class MyClass(object):
    __metaclass__ = MyMeta
```

Practical Example: Enforcing Method Implementation

```
class ABCMeta(type):
    def __new__(mcs, name, bases, attrs):
        # Skip validation for the base class itself
        if name == 'AbstractClass':
            return super().__new__(mcs, name, bases, attrs)

        # Check if required methods are implemented
        if 'required_method' not in attrs:
            raise TypeError(
                f"Class {name} must implement 'required_method'"
            )

        return super().__new__(mcs, name, bases, attrs)

class AbstractClass(metaclass=ABCMeta):
    def required_method(self):
        raise NotImplementedError(
            "Subclasses must implement required_method"
        )

# This will work
class ValidClass(AbstractClass):
    def required_method(self):
        return "Implementation provided"

# This will raise a TypeError at class definition time
class InvalidClass(AbstractClass):
    pass  # Missing required_method implementation
```

Metaclasses are a powerful but complex feature. They're rarely needed in everyday programming but can be valuable tools for framework developers and in scenarios where you need to enforce specific behaviors at the class level rather than the instance level.

Abstract Base Classes (ABCs)

What are ABCs?

Abstract Base Classes are classes that cannot be instantiated directly and are designed to be subclassed. They define an interface that derived classes must implement.

Key Features

- Defined in the abc module
- Cannot be instantiated directly
- Can include abstract and concrete methods
- Enforce that subclasses implement required methods
- Can be used to verify that a class implements a particular interface

Key Components

- ABC class: Base class for defining ABCs
- @abstractmethod: Decorator for abstract methods
- @abstractproperty: Decorator for abstract properties
- @abstractclassmethod: Decorator for abstract class methods
- @abstractstaticmethod: Decorator for abstract static methods

Example: Shape ABC

```
from abc import ABC, abstractmethod

class Shape(ABC):
 @abstractmethod
 def area(self):
 """Calculate the area of the shape."""
 pass

 @abstractmethod
 def perimeter(self):
 """Calculate the perimeter of the shape."""
 pass

 def describe(self):
 """Non-abstract method with default implementation."""
 return f"A shape with area {self.area()} and perimeter
{self.perimeter()}"

# This will work
class Rectangle(Shape):
 def __init__(self, width, height):
 self.width = width
 self.height = height

 def area(self):
 return self.width * self.height

 def perimeter(self):
 return 2 * (self.width + self.height)

# This will raise TypeError when instantiated
class InvalidCircle(Shape):
 def __init__(self, radius):
 self.radius = radius

 # Missing perimeter method
 def area(self):
 return 3.14 * self.radius ** 2
```

ABCs vs. Interfaces

Python's Approach to Interfaces

Python doesn't have a formal interface keyword like Java or C#. Instead, ABCs serve as Python's mechanism for defining interfaces.

Duck Typing vs. Formal Interfaces

- Duck typing: "If it walks like a duck and quacks like a duck, it's a duck"
- Traditional Python relies heavily on duck typing
- ABCs provide more formal interface checking
- ABCs catch errors at instantiation time rather than when methods are called

Benefits of ABCs as Interfaces

- Better documentation of expected behavior
- Earlier error detection
- Type checking support
- Clearer design intentions

Runtime Interface Checking

```
from abc import ABC, abstractmethod

class JSONSerializable(ABC):
    @abstractmethod
    def to_json(self):
        """Return a JSON string representation."""
        pass

    @classmethod
    def __subclasshook__(cls, subclass):
        """Check if a class implements the interface."""
        return (
            hasattr(subclass, 'to_json') and
            callable(subclass.to_json) or
            NotImplemented
        )

# No inheritance, but passes subclasshook
class User:
    def __init__(self, name, email):
        self.name = name
        self.email = email

    def to_json(self):
        return f'{{"name": "{self.name}", "email": "{self.email}"}}'

# Check if User implements JSONSerializable
print(issubclass(User, JSONSerializable))  # True

# Use isinstance for type checking
user = User("John", "john@example.com")
if isinstance(user, JSONSerializable):
    print(user.to_json())
```

ABCs provide a way to formalize interfaces in Python, bringing some of the benefits of statically typed languages while maintaining Python's flexibility. They're particularly useful in larger applications and libraries where you want to ensure consistent behavior across different implementations.

Design Patterns: Introduction

What are Design Patterns?

Design patterns are reusable solutions to common problems in software design. They represent best practices evolved over time by experienced software developers.

Benefits of Design Patterns

- Proven solutions to recurring problems
- Common vocabulary for developers
- Higher-level abstractions than code
- Improved code structure and maintainability
- Faster development through reusable templates
- Reduced bugs through tested approaches

Pattern Categories

- **Creational**: Object creation mechanisms
- **Structural**: Object composition and relationships
- **Behavioral**: Object communication and responsibility

Common Design Patterns in Python

Creational Patterns

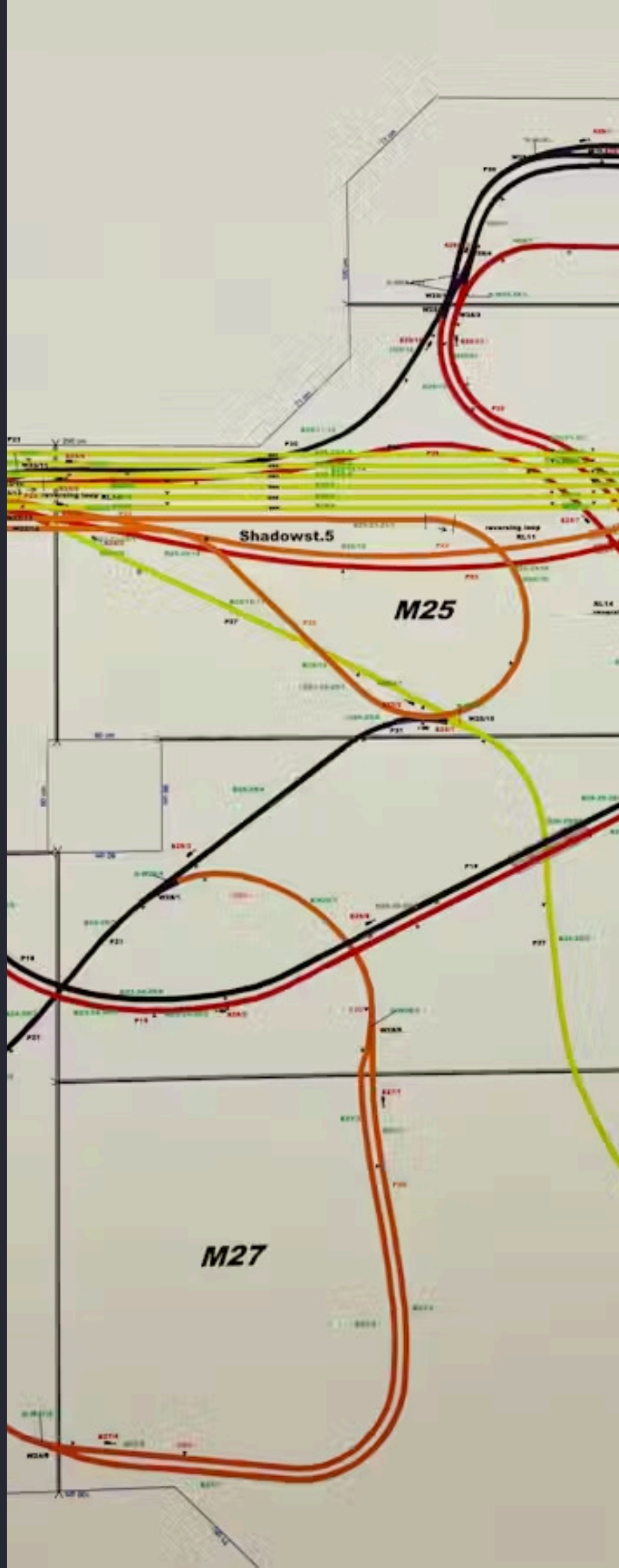
- Factory Method
- Abstract Factory
- Builder
- Singleton
- Prototype

Structural Patterns

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Behavioral Patterns

- Chain of Responsibility
- Command
- Iterator
- Observer
- Strategy
- Template Method
- Visitor



Abstract Factory Pattern

What is the Abstract Factory Pattern?

A creational pattern that provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Key Components

- **Abstract Factory**: Interface with methods for creating abstract products
- **Concrete Factory**: Implements the Abstract Factory to create concrete products
- **Abstract Product**: Interface for a type of product object
- **Concrete Product**: Implements the Abstract Product interface
- **Client**: Uses only interfaces declared by Abstract Factory and Abstract Product

When to Use

- System needs to be independent of how its products are created
- System should be configured with one of multiple families of products
- Family of related objects is designed to be used together
- You want to provide a library of products and reveal only interfaces

```
from abc import ABC, abstractmethod

# Abstract Products
class Button(ABC):
  @abstractmethod
  def paint(self):
    pass

class Checkbox(ABC):
  @abstractmethod
  def paint(self):
    pass

# Concrete Products
class WindowsButton(Button):
  def paint(self):
    return "Rendering a Windows button"

class WindowsCheckbox(Checkbox):
  def paint(self):
    return "Rendering a Windows checkbox"

class MacButton(Button):
  def paint(self):
    return "Rendering a macOS button"

class MacCheckbox(Checkbox):
  def paint(self):
    return "Rendering a macOS checkbox"

# Abstract Factory
class GUIFactory(ABC):
  @abstractmethod
  def create_button(self):
    pass

  @abstractmethod
  def create_checkbox(self):
    pass

# Concrete Factories
class WindowsFactory(GUIFactory):
  def create_button(self):
    return WindowsButton()

  def create_checkbox(self):
    return WindowsCheckbox()

class MacFactory(GUIFactory):
  def create_button(self):
    return MacButton()

  def create_checkbox(self):
    return MacCheckbox()

# Client code
def create_ui(factory):
  button = factory.create_button()
  checkbox = factory.create_checkbox()
  return button.paint(), checkbox.paint()

# Usage
windows_ui = create_ui(WindowsFactory())
mac_ui = create_ui(MacFactory())
```

Façade Pattern

What is the Façade Pattern?

A structural pattern that provides a simplified interface to a complex subsystem of classes, making it easier to use.

Key Components

- **Façade**: Provides a simple interface to complex subsystem
- **Subsystem Classes**: Implement subsystem functionality
- **Client**: Uses the Façade to interact with the subsystem

Benefits

- Simplifies client interface to complex systems
- Decouples client from subsystem implementation details
- Promotes loose coupling between clients and subsystems
- Makes subsystems easier to use and reduces dependencies
- Allows subsystems to evolve without affecting clients

```
# Complex subsystem components
class CPU:
    def freeze(self):
        return "CPU: Freezing processor"

    def jump(self, address):
        return f"CPU: Jumping to address {address}"

    def execute(self):
        return "CPU: Executing commands"

class Memory:
    def load(self, address, data):
        return f"Memory: Loading {data} at address {address}"

class HardDrive:
    def read(self, sector, size):
        return f"HardDrive: Reading {size}KB from sector
{sector}"

# Façade
class ComputerFacade:
    def __init__(self):
        self.cpu = CPU()
        self.memory = Memory()
        self.hard_drive = HardDrive()

    def start(self):
        operations = [
            self.cpu.freeze(),
            self.hard_drive.read(0, 1024),
            self.memory.load(0, "boot data"),
            self.cpu.jump(0),
            self.cpu.execute()
        ]
        return "\n".join(operations)

# Client code
computer = ComputerFacade()
boot_sequence = computer.start()
print(boot_sequence)
```

The Façade pattern is particularly useful when working with complex libraries, frameworks, or legacy code. It provides a simple entry point to a complex subsystem, making the code more readable and reducing dependencies. In Python, this pattern is often used when wrapping complex APIs or when creating simpler interfaces to complex systems.

Chain of Responsibility Pattern

What is the Chain of Responsibility Pattern?

A behavioral pattern that passes a request along a chain of handlers. Each handler decides whether to process the request or pass it to the next handler in the chain.

Key Components

- **Handler**: Interface defining how to handle requests and link to next handler
- **Concrete Handlers**: Implement the Handler interface, process requests or pass them along
- **Client**: Initiates the request to the first handler in the chain

When to Use

- More than one object may handle a request
- Handler isn't known in advance
- Handler should be determined automatically
- Request should be passed to multiple objects
- The set of handlers can be dynamically defined

```
from abc import ABC, abstractmethod

# Handler interface
class Handler(ABC):
  @abstractmethod
  def set_next(self, handler):
    pass

  @abstractmethod
  def handle(self, request):
    pass

# Base handler implementation
class AbstractHandler(Handler):
  _next_handler = None

  def set_next(self, handler):
    self._next_handler = handler
    # Return handler to allow chaining
    return handler

  @abstractmethod
  def handle(self, request):
    if self._next_handler:
      return \
        self._next_handler.handle(request)
    return None

# Concrete handlers
class AuthenticationHandler(AbstractHandler):
  def handle(self, request):
    if "authenticated" in request:
      print("Authentication successful")
      return super().handle(request)
    else:
      return "Authentication failed"

class AuthorizationHandler(AbstractHandler):
  def handle(self, request):
    if "admin" in request:
      print("User has admin rights")
      return super().handle(request)
    else:
      return "User not authorized"

class ValidationHandler(AbstractHandler):
  def handle(self, request):
    if "valid_data" in request:
      print("Data is valid")
      return super().handle(request)
    else:
      return "Invalid data"

# Client code
def client_code(handler):
 requests = [
 {"authenticated": True, "admin": True, "valid_data": True},
 {"authenticated": True, "admin": False, "valid_data": True},
 {"authenticated": False}
 ]

 for request in requests:
   result = handler.handle(request)
   if result:
     print(f"Result: {result}\n")
   else:
     print("Request passed through all handlers\n")

# Set up the chain
auth = AuthenticationHandler()
author = AuthorizationHandler()
valid = ValidationHandler()

auth.set_next(author).set_next(valid)

# Run client code
client_code(auth)
```

Multi-threading vs. Multi-processing

Concurrency in Python

Concurrency is about handling multiple tasks at once. Python offers several approaches to concurrent programming.

Multi-threading

Key Features

- Multiple threads within a single process
- Threads share the same memory space
- Low overhead for creating threads
- Python's Global Interpreter Lock (GIL) limits parallel execution
- Best for I/O-bound tasks (network, file operations)
- Not effective for CPU-bound tasks due to GIL

Implementation with threading

```
import threading
import time

def io_task(name, delay):
    print(f"Task {name} started")
    time.sleep(delay)  # Simulate I/O operation
    print(f"Task {name} completed")

# Create threads
threads = []
for i in range(5):
    t = threading.Thread(target=io_task, args=(i, 1))
    threads.append(t)
    t.start()

# Wait for all threads to complete
for t in threads:
    t.join()

print("All tasks completed")
```

Multi-processing

Key Features

- Multiple independent processes
- Each process has its own Python interpreter and memory space
- Not affected by the GIL
- Higher overhead than threading
- Best for CPU-bound tasks
- Communication between processes is more complex

Implementation with multiprocessing

```
import multiprocessing
import time

def cpu_task(name, numbers):
    print(f"Process {name} started")
    result = sum(n ** 2 for n in numbers)
    print(f"Process {name} result: {result}")
    return result

if __name__ == "__main__":
    # Create processes
    processes = []
    for i in range(4):
        numbers = range(i * 1000000, (i + 1) * 1000000)
        p = multiprocessing.Process(
            target=cpu_task,
            args=(i, numbers)
        )
        processes.append(p)
        p.start()

    # Wait for all processes to complete
    for p in processes:
        p.join()

    print("All processes completed")
```

The Global Interpreter Lock (GIL)

The GIL is a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecode at once. This means that even with multiple threads, Python can only execute one thread at a time, limiting the performance benefits of multi-threading for CPU-bound tasks.

When to Use Threading

- I/O-bound tasks: network operations, file reading/writing, database queries
- GUI applications to keep the interface responsive
- When you need to maintain shared state with minimal overhead
- When tasks involve a lot of waiting for external resources

When to Use Multiprocessing

- CPU-bound tasks: data processing, numerical computations
- When you need to utilize multiple CPU cores
- When tasks are relatively independent and don't require much shared state
- When the overhead of process creation is acceptable compared to the computation time

Asynchronous Programming: Coroutines with asyncio

What are Coroutines?

Coroutines are special functions that can be paused and resumed, allowing concurrent execution without using threads or processes. They're particularly effective for I/O-bound tasks.

Key Concepts

- Cooperative multitasking (tasks yield control)
- Single-threaded, event-driven architecture
- Non-blocking I/O operations
- Defined using async def and called with await
- Managed by an event loop

Basic asyncio Structure

```
import asyncio

async def main():
    # Your async code here
    await asyncio.sleep(1)
    print("Hello, async world!")

# Run the event loop
asyncio.run(main())
```

Key asyncio Components

- **async def**: Defines a coroutine function
- **await**: Pauses execution until the awaited coroutine completes
- **asyncio.run()**: Runs a coroutine and manages the event loop
- **asyncio.create_task()**: Schedules a coroutine to run concurrently
- **asyncio.gather()**: Runs multiple coroutines concurrently
- **asyncio.sleep()**: Non-blocking sleep function

Running Tasks Concurrently

```
import asyncio

async def fetch_data(name, delay):
    print(f"Start fetching {name}")
    await asyncio.sleep(delay)  # Simulate network request
    print(f"Done fetching {name}")
    return f"{name} result"

async def main():
    # Run tasks concurrently
    results = await asyncio.gather(
        fetch_data("API 1", 2),
        fetch_data("API 2", 1),
        fetch_data("API 3", 3)
    )
    print(results)

asyncio.run(main())
```

Practical Example: Asynchronous Web Scraper

```
import asyncio
import aiohttp  # pip install aiohttp
import time

async def fetch_url(session, url):
    """Fetch content from a URL asynchronously."""
    print(f"Fetching: {url}")
    try:
        async with session.get(url) as response:
            return await response.text()
    except Exception as e:
        print(f"Error fetching {url}: {e}")
        return None

async def process_urls(urls):
    """Process multiple URLs concurrently."""
    start_time = time.time()

    # Create a shared session for all requests
    async with aiohttp.ClientSession() as session:
        # Create tasks for all URLs
        tasks = [fetch_url(session, url) for url in urls]

        # Run all tasks concurrently and get results
        results = await asyncio.gather(*tasks)

        # Process results
        for url, html in zip(urls, results):
            if html:
                print(f"Processed {url}: {len(html)} characters")
            else:
                print(f"Failed to process {url}")

    duration = time.time() - start_time
    print(f"Processed {len(urls)} URLs in {duration:.2f} seconds")

# Sample URLs to fetch
sample_urls = [
    "https://www.python.org",
    "https://www.github.com",
    "https://www.stackoverflow.com",
    "https://www.bbc.co.uk",
    "https://www.wikipedia.org"
]

# Run the async function
asyncio.run(process_urls(sample_urls))
```

Asynchronous programming with coroutines provides a powerful way to handle concurrent operations, especially I/O-bound tasks like network requests, without the overhead of threads or processes. It offers an elegant, single-threaded solution for high-concurrency applications.

Advanced Concurrency Patterns

Thread Pools with concurrent.futures

A high-level interface for asynchronously executing callables.

```
from concurrent.futures import ThreadPoolExecutor
import requests
import time

def fetch_url(url):
    """Fetch the content of a URL."""
    try:
        response = requests.get(url)
        return f"{url}: {len(response.text)} bytes"
    except Exception as e:
        return f"{url}: Error - {str(e)}"

urls = [
    "https://www.python.org",
    "https://www.github.com",
    "https://www.stackoverflow.com",
    "https://www.bbc.co.uk"
]

# Using ThreadPoolExecutor for concurrent HTTP requests
start = time.time()

with ThreadPoolExecutor(max_workers=4) as executor:
    # Submit tasks and get Future objects
    future_to_url = {
        executor.submit(fetch_url, url): url for url in urls
    }

    # Process results as they complete
    for future in
concurrent.futures.as_completed(future_to_url):
        url = future_to_url[future]
        try:
            result = future.result()
            print(result)
        except Exception as e:
            print(f"{url} generated an exception: {e}")

print(f"Time taken: {time.time() - start:.2f} seconds")
```

Process Pools with concurrent.futures

Similar interface to ThreadPoolExecutor but uses processes instead of threads.

```
from concurrent.futures import ProcessPoolExecutor
import time
import math

def is_prime(n):
    """Check if a number is prime."""
    if n < 2:
        return False
    for i in range(2, int(math.sqrt(n)) + 1):
        if n % i == 0:
            return False
    return True

def count_primes(start, end):
    """Count prime numbers in a range."""
    return sum(1 for n in range(start, end) if is_prime(n))

# Define ranges to check for primes
ranges = [(10**6, 2*10**6), (2*10**6, 3*10**6),
          (3*10**6, 4*10**6), (4*10**6, 5*10**6)]

start = time.time()

# Use ProcessPoolExecutor for CPU-bound tasks
with ProcessPoolExecutor(max_workers=4) as executor:
    results = list(executor.map(
        lambda r: count_primes(r[0], r[1]), ranges
    ))

total_primes = sum(results)
print(f"Found {total_primes} prime numbers")
print(f"Time taken: {time.time() - start:.2f} seconds")
```

Asynchronous Context Managers

```
import asyncio
import aiohttp

class AsyncSession:
  """Async context manager for HTTP session."""
  def __init__(self):
    self.session = None

  async def __aenter__(self):
    self.session = aiohttp.ClientSession()
    return self.session

  async def __aexit__(self, exc_type, exc_val, exc_tb):
    if self.session:
      await self.session.close()

  async def fetch_urls(urls):
    results = []

    # Use async context manager
    async with AsyncSession() as session:
      # Create tasks
      tasks = []
      for url in urls:
        tasks.append(fetch_url(session, url))

      # Execute all tasks concurrently
      results = await asyncio.gather(*tasks)

    return results

  async def fetch_url(session, url):
    async with session.get(url) as response:
      return await response.text()

# Usage
urls = ["https://www.python.org", "https://www.github.com"]
results = asyncio.run(fetch_urls(urls))
```

File Handling in Python

Basic File Operations

Opening Files

```
# Basic syntax
file = open(filename, mode)

# Common modes
# 'r' - Read (default)
# 'w' - Write (creates/truncates)
# 'a' - Append
# 'b' - Binary mode
# 't' - Text mode (default)
# '+' - Update (read & write)

# Examples
file = open('data.txt', 'r')      # Read text
file = open('data.bin', 'rb')     # Read binary
file = open('output.txt', 'w')    # Write text
file = open('output.bin', 'wb')   # Write binary
file = open('log.txt', 'a')       # Append text
```

Closing Files

```
# Always close files when done
file.close()

# Better: use with statement (context manager)
with open('data.txt', 'r') as file:
    # File operations here
    data = file.read()
# File is automatically closed when exiting the block
```

Reading Files

```
# Read entire file content
with open('data.txt', 'r') as file:
    content = file.read()

# Read line by line
with open('data.txt', 'r') as file:
    for line in file:
        print(line.strip())  # strip() removes newline

# Read all lines into a list
with open('data.txt', 'r') as file:
    lines = file.readlines()

# Read specific amount of data
with open('data.txt', 'r') as file:
    chunk = file.read(1024)  # Read 1024 bytes
```

Writing Files

```
# Write string to file
with open('output.txt', 'w') as file:
    file.write('Hello, world!\n')
    file.write('Another line.\n')

# Write multiple lines at once
lines = ['Line 1', 'Line 2', 'Line 3']
with open('output.txt', 'w') as file:
    file.writelines(line + '\n' for line in lines)
```

File Positions and Seeking

```
with open('data.txt', 'r') as file:
    # Get current position
    position = file.tell()

    # Read first 5 bytes
    data = file.read(5)

    # Move to specific position (from start)
    file.seek(0)  # Go back to start

    # Move relative to current position
    file.seek(10, 1)  # Move 10 bytes forward from current position

    # Move relative to end
    file.seek(-20, 2)  # Move 20 bytes before the end of file
```

Working with Different File Types

```
# CSV Files
import csv

# Reading CSV
with open('data.csv', 'r', newline='') as file:
 reader = csv.reader(file)
 for row in reader:
 print(row) # row is a list of values

# Writing CSV
with open('output.csv', 'w', newline='') as file:
 writer = csv.writer(file)
 writer.writerow(['Name', 'Age', 'City'])
 writer.writerow(['Alice', 30, 'London'])
 writer.writerow(['Bob', 25, 'Manchester'])

# JSON Files
import json

# Reading JSON
with open('data.json', 'r') as file:
 data = json.load(file) # Parse JSON into Python objects

# Writing JSON
with open('output.json', 'w') as file:
 json.dump(data, file, indent=4) # Write Python objects as JSON
```

More File Handling Techniques

Working with Paths

```
import os
from pathlib import Path

# Using os.path
current_dir = os.getcwd()
file_path = os.path.join(current_dir, 'data', 'file.txt')
file_exists = os.path.exists(file_path)
file_size = os.path.getsize(file_path)
is_file = os.path.isfile(file_path)
is_dir = os.path.isdir(file_path)

# Using pathlib (more modern, object-oriented)
path = Path('data/file.txt')
path = Path.home() / 'data' / 'file.txt'  # Path joining
path.exists()
path.is_file()
path.is_dir()
path.stat().st_size  # File size
path.parent  # Parent directory
path.name  # File name with extension
path.stem  # File name without extension
path.suffix  # File extension
```

Temporary Files

```
import tempfile

# Temporary file that's automatically cleaned up
with tempfile.TemporaryFile() as temp:
    temp.write(b'Writing to temporary file')
    temp.seek(0)
    data = temp.read()

# Named temporary file
with tempfile.NamedTemporaryFile() as temp:
    print(f"Temp file created: {temp.name}")
    temp.write(b'Named temporary file')
    temp.flush()  # Ensure data is written to disk

# Temporary directory
with tempfile.TemporaryDirectory() as temp_dir:
    print(f"Temp directory created: {temp_dir}")
    # Use temp_dir for temporary file storage
```

Working with Binary Files

```
# Reading binary data
with open('image.png', 'rb') as file:
    binary_data = file.read()
    # Process binary data

# Writing binary data
with open('output.bin', 'wb') as file:
    file.write(binary_data)

# Working with structured binary data
import struct

# Pack values into binary format
# Format: 'I' = unsigned int, 'f' = float, '10s' = 10-char string
packed = struct.pack('If10s', 123, 45.67, b'Hello')

# Write packed data to file
with open('data.bin', 'wb') as file:
    file.write(packed)

# Read and unpack binary data
with open('data.bin', 'rb') as file:
    data = file.read()
    unpacked = struct.unpack('If10s', data)
    print(unpacked)  # (123, 45.67, b'Hello\x00\x00\x00\x00\x00')
```

Memory-Mapped Files

```
import mmap

# Memory-map a file (useful for large files)
with open('large_file.bin', 'rb') as file:
 # Map the file into memory
 with mmap.mmap(file.fileno(), 0, access=mmap.ACCESS_READ) as mm:
 # Use mm like a file or bytes object
 print(len(mm)) # File size

 # Random access is efficient
 mm.seek(1000000)
 chunk = mm.read(1024)

 # Search in the file
 position = mm.find(b'search_term')
 if position != -1:
 print(f"Found at position: {position}")
```

File Handling Best Practices

Error Handling

```
# Always use try/except for file operations
try:
    with open('data.txt', 'r') as file:
        content = file.read()
except FileNotFoundError:
    print("File doesn't exist")
except PermissionError:
    print("No permission to read file")
except IsADirectoryError:
    print("Expected a file, got a directory")
except IOError as e:
    print(f"I/O error: {e}")
```

Working with Large Files

```
# Process large files line by line
def process_large_file(filename):
    with open(filename, 'r') as file:
        for line in file:  # Memory-efficient iteration
            yield line.strip()

# Or chunk by chunk
def read_in_chunks(file_path, chunk_size=1024*1024):
    """Read a file in chunks of specified size."""
    with open(file_path, 'r') as file:
        while True:
            chunk = file.read(chunk_size)
            if not chunk:
                break
            yield chunk
```

File Locking

```
import fcntl  # Unix-based systems only

def lock_file(file_obj, exclusive=True):
    """Lock a file to prevent concurrent access."""
    lock_type = fcntl.LOCK_EX if exclusive else fcntl.LOCK_SH
    fcntl.flock(file_obj.fileno(), lock_type)

def unlock_file(file_obj):
    """Release a file lock."""
    fcntl.flock(file_obj.fileno(), fcntl.LOCK_UN)

# Usage
with open('data.txt', 'w') as file:
    try:
        lock_file(file)  # Exclusive lock for writing
        file.write('Critical data')
    finally:
        unlock_file(file)
```

Configuration Files

```
# INI files
import configparser

# Reading config
config = configparser.ConfigParser()
config.read('config.ini')
db_host = config['database']['host']
db_port = config.getint('database', 'port')

# Writing config
config['database'] = {
    'host': 'localhost',
    'port': '5432',
    'user': 'admin'
}
with open('config.ini', 'w') as file:
    config.write(file)
```

File System Operations

```
import os
import shutil
from pathlib import Path

# Create directory
os.makedirs('data/output', exist_ok=True)  # Create parent directories if needed

# List directory contents
files = os.listdir('data')
with_path = [os.path.join('data', f) for f in files]

# More powerful with pathlib
data_dir = Path('data')
files = list(data_dir.glob('*.txt'))  # List all .txt files
recursive_files = list(data_dir.rglob('*.txt'))  # Recursive search

# Copy files
shutil.copy('source.txt', 'destination.txt')  # Copy file
shutil.copytree('source_dir', 'dest_dir')  # Copy directory recursively

# Move/rename files
os.rename('old_name.txt', 'new_name.txt')
shutil.move('source.txt', 'destination/source.txt')

# Delete files
os.remove('file.txt')  # Delete file
os.rmdir('empty_dir')  # Delete empty directory
shutil.rmtree('directory')  # Delete directory and all contents (use with caution!)
```

Effective file handling is essential for many Python applications. Always use context managers (with statement) to ensure files are properly closed, handle errors appropriately, and consider memory usage when working with large files.

Data Processing Workflows

Creating Data Pipelines

Combining comprehensions, generators, and functions to process data efficiently.

```
import csv
import json
from datetime import datetime

# Sample data processing pipeline
def process_sales_data(csv_path):
    # Read and parse CSV
    with open(csv_path, 'r') as file:
        reader = csv.DictReader(file)
        sales = list(reader)

    # Clean and transform data
    sales_cleaned = [
        {
            "product": row["Product"],
            "quantity": int(row["Quantity"]),
            "price": float(row["Price"]),
            "date": datetime.strptime(
                row["Date"], "%Y-%m-%d"
            ).date(),
            "total": int(row["Quantity"]) * float(row["Price"])
        }
        for row in sales
    ]

    # Filter to recent sales
    recent_sales = [
        sale for sale in sales_cleaned
        if (datetime.now().date() - sale["date"]).days < 30
    ]

    # Calculate statistics
    total_revenue = sum(sale["total"] for sale in recent_sales)
    avg_sale = total_revenue / len(recent_sales) if recent_sales
else 0
    product_counts = {}

    for sale in recent_sales:
        product = sale["product"]
        product_counts[product] = product_counts.get(product,
0) + 1

    top_products = sorted(
        product_counts.items(),
        key=lambda x: x[1],
        reverse=True
    )[:5]

    # Create report
    report = {
        "total_revenue": total_revenue,
        "average_sale": avg_sale,
        "top_products": top_products,
        "total_sales": len(recent_sales)
    }

    # Write results
    with open('sales_report.json', 'w') as outfile:
        json.dump(report, outfile, indent=4, default=str)

    return report
```

Memory-Efficient Data Processing

Using generators for large datasets.

```
def process_large_log_file(log_path, pattern):
    """Process a large log file looking for a pattern."""
    def read_in_chunks(file_obj, chunk_size=1024*1024):
        """Read file in chunks to avoid loading it all into
memory."""
        while True:
            data = file_obj.read(chunk_size)
            if not data:
                break
            yield data

    def extract_lines(chunks):
        """Extract lines from chunks, handling partial lines."""
        buffer = ""
        for chunk in chunks:
            buffer += chunk
            lines = buffer.split('\n')
            # Keep the last (potentially partial) line in buffer
            buffer = lines.pop()
            for line in lines:
                yield line
        if buffer:
            yield buffer

    def filter_lines(lines, pattern):
        """Filter lines that contain the pattern."""
        for line in lines:
            if pattern in line:
                yield line

    def parse_timestamp(line):
        """Extract and parse timestamp from a log line."""
        try:
            timestamp_str = line.split('[')[1].split(']')[0]
            return datetime.strptime(timestamp_str, "%Y-%m-%d
%H:%M:%S")
        except (IndexError, ValueError):
            return None

    # Create the processing pipeline
    with open(log_path, 'r') as log_file:
        chunks = read_in_chunks(log_file)
        lines = extract_lines(chunks)
        matching_lines = filter_lines(lines, pattern)

        # Count matches by hour
        hour_counts = {}
        for line in matching_lines:
            timestamp = parse_timestamp(line)
            if timestamp:
                hour = timestamp.replace(minute=0, second=0)
                hour_counts[hour] = hour_counts.get(hour, 0) + 1

        return hour_counts
```

Parallel Data Processing

```
from concurrent.futures import ProcessPoolExecutor
import pandas as pd
import numpy as np
import os

def process_chunk(chunk_file):
 """Process a single data chunk."""
 # Read the chunk
 df = pd.read_csv(chunk_file)

 # Perform computations
 result = {
 'file': os.path.basename(chunk_file),
 'count': len(df),
 'mean': df['value'].mean(),
 'std': df['value'].std(),
 'min': df['value'].min(),
 'max': df['value'].max()
 }

 return result

def split_large_csv(input_file, output_dir, chunk_size=100000):
 """Split a large CSV file into smaller chunks."""
 if not os.path.exists(output_dir):
 os.makedirs(output_dir)

 # Read and write in chunks
 chunk_files = []
 for i, chunk in enumerate(pd.read_csv(input_file, chunksize=chunk_size)):
 chunk_file = os.path.join(output_dir, f"chunk_{i}.csv")
 chunk.to_csv(chunk_file, index=False)
 chunk_files.append(chunk_file)

 return chunk_files

def parallel_process_large_csv(input_file, output_dir="./chunks", max_workers=4):
 """Process a large CSV file in parallel."""
 # Split the file into chunks
 chunk_files = split_large_csv(input_file, output_dir)

 # Process chunks in parallel
 results = []
 with ProcessPoolExecutor(max_workers=max_workers) as executor:
 for result in executor.map(process_chunk, chunk_files):
 results.append(result)

 # Combine results
 combined = {
 'total_count': sum(r['count'] for r in results),
 'chunks_processed': len(results),
 'chunk_details': results
 }

 return combined
```

Next Steps: Expanding Your Python Knowledge

1

Advanced Libraries

Explore specialized libraries like NumPy, pandas, scikit-learn, and Django to solve real-world problems in data science, machine learning, and web development.

2

Testing and CI/CD

Master pytest, unittest, and continuous integration tools to build robust, reliable applications with automated testing pipelines.

3

Performance Optimization

Learn techniques for profiling and optimizing Python code, including Cython, Numba, and PyPy for high-performance computing.

4

Design Patterns

Deepen your understanding of software architecture by implementing more design patterns and studying their applications in large-scale projects.

Recommended Resources

- **Books**: "Fluent Python" by Luciano Ramalho, "Python Cookbook" by David Beazley and Brian K. Jones
- **Documentation**: Python official documentation, especially the language reference and library reference
- **Open Source Projects**: Contributing to open-source Python projects is an excellent way to learn from experienced developers

Thank you for joining us for this intensive exploration of advanced Python concepts. The journey doesn't end here—Python's ecosystem is vast and constantly evolving. Keep exploring, keep coding, and most importantly, keep solving interesting problems!

Chandrashekar Babu <training@chandrashekar.info> | <chandra@slashprog.com> | www.chandrashekar.info | www.slashprog.com

YouTube: