Intermediate Python Programming

Taking Your Python Skills to the Next Level

Course Overview

- **Object-Oriented Programming (OOP)** - Classes, inheritance, ABC, Protocol, mixins, operator overloading
- **Functional Programming** - Lambda, map, filter, reduce, generator pipelines, coroutines
- **Decorators & Generators** - Advanced function features, itertools
- **Data Handling** - File I/O, JSON, CSV, error handling
- **Standard Library** - collections, os, sys modules
- **Testing & Packaging** - Unit tests, modules, packages

What is Object-Oriented Programming?

OOP is a programming paradigm based on the concept of "objects" that contain data and code.

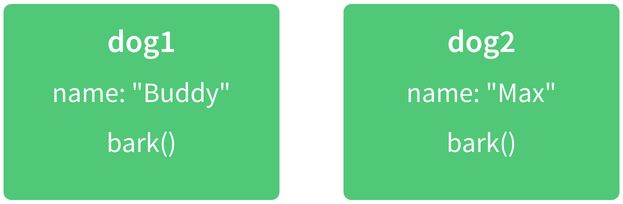
> ### The Four Pillars of OOP
>
> - **Encapsulation** - Bundling data and methods together
> - **Abstraction** - Hiding complex implementation details
> - **Inheritance** - Creating new classes from existing ones
> - **Polymorphism** - Using a unified interface for different types

Classes and Objects

Class: A blueprint for creating objects

Object: An instance of a class

```python
# Define a class
class Dog:
    def __init__(self, name):
        self.name = name

    def bark(self):
        return f"{self.name} says Woof!"

# Create objects (instances)
dog1 = Dog("Buddy")
dog2 = Dog("Max")

print(dog1.bark())  # Buddy says Woof!
print(dog2.bark())  # Max says Woof!
```



dog1

name: "Buddy"

bark()

dog2

name: "Max"

bark()

Understanding __init__ and self

> **__init__:** The constructor method - called when an object is created
> **self:** Refers to the instance itself

```python
class Person:
    def __init__(self, name, age):
        # self.attribute = parameter
        self.name = name  # Instance attribute
        self.age = age
        self.greeting = f"Hello, I'm {name}!"

    def introduce(self):
        # self is automatically passed as first parameter
        return f"{self.greeting} I'm {self.age} years old."

    def have_birthday(self):
        self.age += 1  # Modify instance attribute
        return f"Happy Birthday! {self.name} is now {self.age}!"

# Create instance
person = Person("Alice", 30)
print(person.introduce())      # Hello, I'm Alice! I'm 30 years old.
print(person.have_birthday())  # Happy Birthday! Alice is now 31!
```

Instance vs Class Attributes

```
class BankAccount:
    # Class attribute - shared by ALL instances
    bank_name = "Python Bank"
    interest_rate = 0.02
    total_accounts = 0

    def __init__(self, owner, balance):
        # Instance attributes - unique to each instance
        self.owner = owner
        self.balance = balance
        BankAccount.total_accounts += 1

    def deposit(self, amount):
        self.balance += amount
        return self.balance

# Each instance has its own owner and balance
account1 = BankAccount("Alice", 1000)
account2 = BankAccount("Bob", 2000)

print(account1.bank_name)         # Python Bank (from class)
print(account1.balance)           # 1000 (instance-specific)
print(account2.balance)           # 2000 (instance-specific)
print(BankAccount.total_accounts)  # 2 (class attribute)
```

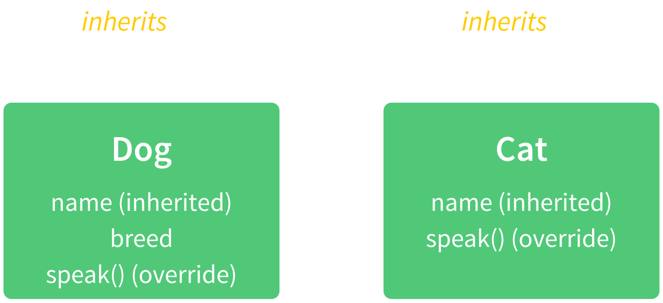
Types of Methods

```python
class MyClass:
    class_var = "I'm a class variable"

    def __init__(self, value):
        self.instance_var = value

    # Instance method - operates on instance data
    def instance_method(self):
        return f"Instance: {self.instance_var}"

    # Class method - operates on class data
    @classmethod
    def class_method(cls):
        return f"Class: {cls.class_var}"

    # Static method - doesn't access instance or class data
    @staticmethod
    def static_method(x, y):
        return f"Static: {x + y}"

# Usage
obj = MyClass("hello")
print(obj.instance_method())       # Instance: hello
print(MyClass.class_method())      # Class: I'm a class variable
print(MyClass.static_method(5, 3)) # Static: 8
```

Inheritance

Creating new classes based on existing ones - promotes code reuse

```python
# Parent class (Base class)
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return "Some sound"

# Child class (Derived class)
class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)  # Call parent __init__
        self.breed = breed

    def speak(self):  # Override parent method
        return f"{self.name} says Woof!"

class Cat(Animal):
    def speak(self):
        return f"{self.name} says Meow!"

# Usage
dog = Dog("Buddy", "Golden Retriever")
cat = Cat("Whiskers")
print(dog.speak())  # Buddy says Woof!
print(cat.speak())  # Whiskers says Meow!
```



inherits

inherits

Dog

name (inherited)

breed

speak() (override)

Cat

name (inherited)

speak() (override)

Method Overriding

Child classes can override parent methods to provide specific implementations

```python
class Vehicle:
    def __init__(self, brand):
        self.brand = brand

    def start(self):
        return "Vehicle is starting..."

    def info(self):
        return f"This is a {self.brand} vehicle"

class ElectricCar(Vehicle):
    def __init__(self, brand, battery_capacity):
        super().__init__(brand)
        self.battery_capacity = battery_capacity

    # Override start method
    def start(self):
        return f"{self.brand} electric motor starting silently..."

    # Override info and extend it
    def info(self):
        base_info = super().info()  # Call parent method
        return f"{base_info} with {self.battery_capacity}kWh battery"

car = ElectricCar("Tesla", 75)
print(car.start())  # Tesla electric motor starting silently...
print(car.info())   # This is a Tesla vehicle with 75kWh battery
```

Multiple Inheritance and MRO

A class can inherit from multiple parent classes

```python
class Flyer:
    def fly(self):
        return "Flying through the air"

class Swimmer:
    def swim(self):
        return "Swimming in water"

class Duck(Flyer, Swimmer):  # Multiple inheritance
    def quack(self):
        return "Quack quack!"

# Duck inherits from both Flyer and Swimmer
duck = Duck()
print(duck.fly())   # Flying through the air
print(duck.swim())  # Swimming in water
print(duck.quack()) # Quack quack!

# Method Resolution Order (MRO) - order Python searches for methods
print(Duck.__mro__)
# (<class 'Duck'>, <class 'Flyer'>, <class 'Swimmer'>, <class 'object'>)
```

> **MRO:** Python uses C3 linearization to determine the order in which base classes are searched when looking for a method.

Polymorphism

Different classes can be used interchangeably through a common interface

```python
class Shape:
    def area(self):
        pass

    def perimeter(self):
        pass

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)

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14159 * self.radius ** 2

    def perimeter(self):
        return 2 * 3.14159 * self.radius

# Polymorphism in action - same interface, different implementations
shapes = [Rectangle(5, 3), Circle(4), Rectangle(2, 8)]

for shape in shapes:
    print(f"Area: {shape.area():.2f}, Perimeter: {shape.perimeter():.2f}")
```

Encapsulation: Public, Protected, Private

Control access to class members

```python
class BankAccount:
    def __init__(self, account_number, balance):
        self.account_number = account_number    # Public
        self._balance = balance                 # Protected (convention)
        self.__pin = "1234"                     # Private (name mangling)

    # Public method
    def get_balance(self):
        return self._balance

    # Protected method (convention - can still be accessed)
    def _calculate_interest(self):
        return self._balance * 0.02

    # Private method (name mangled to _BankAccount__validate_pin)
    def __validate_pin(self, pin):
        return pin == self.__pin

    def withdraw(self, amount, pin):
        if self.__validate_pin(pin):
            if amount <= self._balance:
                self._balance -= amount
                return f"Withdrew ${amount}"
            return "Insufficient funds"
        return "Invalid PIN"

account = BankAccount("123456", 1000)
print(account.account_number)      # 123456 (accessible)
print(account._balance)            # 1000 (accessible but discouraged)
# print(account.__pin)             # AttributeError
```

```
(private)
print(account.withdraw(100, "1234"))  # Withdrew $100
```

Properties: Pythonic Getters and Setters

Use @property to create managed attributes with validation

```python
class Temperature:
    def __init__(self, celsius=0):
        self._celsius = celsius  # Private storage

    @property
    def celsius(self):
        """Getter for celsius"""
        return self._celsius

    @celsius.setter
    def celsius(self, value):
        """Setter with validation"""
        if value < -273.15:
            raise ValueError("Temperature below absolute zero!")
        self._celsius = value

    @property
    def fahrenheit(self):
        """Computed property (read-only)"""
        return (self._celsius * 9/5) + 32

    @fahrenheit.setter
    def fahrenheit(self, value):
        """Set celsius from fahrenheit"""
        self._celsius = (value - 32) * 5/9

# Usage - looks like attribute access!
temp = Temperature(25)
print(temp.celsius)     # 25 (uses getter)
print(temp.fahrenheit)  # 77.0 (computed)

temp.fahrenheit = 100   # Uses setter
print(temp.celsius)     # 37.78

temp.celsius = -300     # ValueError: below absolute zero!
```

Properties: Advanced Patterns

COMPUTED PROPERTIES

```
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, value):
        if value < 0:
            raise ValueError("Radius must be
positive")
        self._radius = value

    @property
    def area(self):
        """Computed from radius"""
        return 3.14159 * self._radius ** 2

    @property
    def diameter(self):
        return self._radius * 2

c = Circle(5)
print(c.area)       # 78.54
print(c.diameter)   # 10
```

DELETER PROPERTY

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

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
        if not value.strip():
            raise ValueError("Name cannot be empty")
        self._name = value.strip().title()

    @name.deleter
    def name(self):
        print("Deleting name...")
        self._name = None

p = Person("  john doe  ")
print(p.name)   # John Doe (cleaned)
del p.name      # Deleting name...
print(p.name)   # None
```

Descriptors: The Magic Behind Properties

Control attribute access at the class level

```python
# Descriptor Protocol: __get__, __set__, __delete__

class Validator:
    """A descriptor that validates values"""
    def __init__(self, min_value=None, max_value=None):
        self.min_value = min_value
        self.max_value = max_value

    def __set_name__(self, owner, name):
        self.name = name  # Automatically get attribute name

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        return obj.__dict__.get(self.name)

    def __set__(self, obj, value):
        if self.min_value is not None and value < self.min_value:
            raise ValueError(f"{self.name} must be >= {self.min_value}")
        if self.max_value is not None and value > self.max_value:
            raise ValueError(f"{self.name} must be <= {self.max_value}")
        obj.__dict__[self.name] = value

# Usage - reusable validation!
class Product:
    price = Validator(min_value=0)        # Can't be negative
    quantity = Validator(min_value=0, max_value=1000)

p = Product()
p.price = 29.99      # OK
```

```
p.quantity = 50      # OK
p.price = -10        # ValueError: price must be >= 0
```

Descriptors: Types and Use Cases

DATA DESCRIPTOR (HAS __SET__)

```
class TypedAttribute:
    """Enforce type at assignment"""
    def __init__(self, expected_type):
        self.expected_type = expected_type

    def __set_name__(self, owner, name):
        self.name = name

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        return obj.__dict__.get(self.name)

    def __set__(self, obj, value):
        if not isinstance(value, self.expected_type):
            raise TypeError(
                f"{self.name} must be
{self.expected_type.__name__}"
            )
        obj.__dict__[self.name] = value

class Person:
    name = TypedAttribute(str)
    age = TypedAttribute(int)

p = Person()
p.name = "Alice"  # OK
p.age = "thirty"  # TypeError!
```

NON-DATA DESCRIPTOR (ONLY __GET__)

```
class LazyProperty:
    """Compute once, then cache"""
    def __init__(self, func):
        self.func = func

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        # Compute and cache in instance __dict__
        value = self.func(obj)
        obj.__dict__[self.func.__name__] = value
        return value

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

    @LazyProperty
    def statistics(self):
        """Expensive computation - only done once"""
        print("Computing statistics...")
        return {
            'mean': sum(self.data) / len(self.data),
            'max': max(self.data),
            'min': min(self.data)
        }

analyzer = DataAnalyzer([1, 2, 3, 4, 5])
print(analyzer.statistics)  # Computing...
print(analyzer.statistics)  # Cached!
```

Pattern Matching (Python 3.10+)

Structural pattern matching with the `match` statement

```
# Basic matching - like switch/case but more powerful
def http_status(status):
    match status:
        case 200:
            return "OK"
        case 404:
            return "Not Found"
        case 500:
            return "Internal Server Error"
        case _:  # Default case (wildcard)
            return "Unknown status"

print(http_status(200))  # OK
print(http_status(418))  # Unknown status

# Matching with OR patterns
def classify_day(day):
    match day.lower():
        case "saturday" | "sunday":
            return "Weekend!"
        case "monday" | "tuesday" | "wednesday" |
"thursday" | "friday":
            return "Weekday"
        case _:
            return "Invalid day"

# Matching with guards (if conditions)
def describe_number(n):
    match n:
        case x if x < 0:
            return "Negative"
        case 0:
            return "Zero"
        case x if x > 100:
            return "Large positive"
        case _:
            return "Small positive"
```

Pattern Matching: Structural Patterns

```python
# Matching sequences (lists, tuples)
def process_command(command):
    match command.split():
        case ["quit"]:
            return "Exiting..."
        case ["load", filename]:
            return f"Loading {filename}"
        case ["save", filename, "as", format]:
            return f"Saving {filename} as {format}"
        case ["move", *coordinates]:  # Capture rest
            return f"Moving to {coordinates}"
        case _:
            return "Unknown command"

print(process_command("load data.txt"))       #
Loading data.txt
print(process_command("save doc as pdf"))     # Saving
doc as pdf
print(process_command("move 10 20 30"))       # Moving
to ['10', '20', '30']

# Matching dictionaries
def process_event(event):
    match event:
        case {"type": "click", "x": x, "y": y}:
            return f"Click at ({x}, {y})"
        case {"type": "keypress", "key": key}:
            return f"Key pressed: {key}"
        case {"type": "scroll", "direction": d,
**rest}:
            return f"Scrolling {d}"
        case _:
            return "Unknown event"

# Matching class instances
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def describe_point(point):
    match point:
        case Point(x=0, y=0):
            return "Origin"
        case Point(x=0, y=y):
            return f"On Y-axis at y={y}"
        case Point(x=x, y=0):
            return f"On X-axis at x={x}"
        case Point(x=x, y=y) if x == y:
            return f"On diagonal at ({x}, {y})"
        case _:
            return f"Point at ({point.x}, {point.y})"
```

Interfaces in Python: ABC, Protocol & Mixins

Python's approaches to defining contracts and sharing behavior

> ### THREE COMPLEMENTARY APPROACHES
>
> - **ABC (Abstract Base Classes)** - Formal interfaces with enforcement
> - **Protocol** - Structural subtyping (duck typing with type hints)
> - **Mixins** - Reusable behavior via multiple inheritance

Abstract Base Classes (ABC)

Formal interfaces with runtime enforcement

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    """Abstract base class defining the interface for shapes"""

    @abstractmethod
    def area(self) -> float:
        """Calculate the area - MUST be implemented"""
        pass

    @abstractmethod
    def perimeter(self) -> float:
        """Calculate the perimeter - MUST be implemented"""
        pass

    def describe(self) -> str:
        """Concrete method - shared implementation"""
        return f"{self.__class__.__name__}: area={self.area():.2f}"

class Circle(Shape):
    def __init__(self, radius: float):
        self.radius = radius

    def area(self) -> float:
        return 3.14159 * self.radius ** 2

    def perimeter(self) -> float:
        return 2 * 3.14159 * self.radius

# shape = Shape()  # TypeError: Can't instantiate abstract class
circle = Circle(5)
print(circle.describe())  # Circle: area=78.54
```

Protocol Classes (Structural Subtyping)

Duck typing with type checker support - Python 3.8+

```python
from typing import Protocol

class Drawable(Protocol):
    """Protocol: Any class with these methods is considered Drawable"""

    def draw(self) -> str:
        ...  # No implementation needed

    def get_color(self) -> str:
        ...

class Button:
    """Implicitly implements Drawable - no inheritance needed!"""
    def draw(self) -> str:
        return f"Drawing {self.get_color()} button"

    def get_color(self) -> str:
        return "blue"

    def click(self) -> None:
        print("Clicked!")

def render(item: Drawable) -> None:
    """Works with ANY object that has draw() and get_color()"""
    print(item.draw())

button = Button()
render(button)  # Works! Button matches the Protocol structure
```

ABC vs Protocol: When to Use Which?

Subtyping	Nominal (explicit inheritance)	Structural (duck typing)
Enforcement	Runtime error on instantiation	Type checker only (mypy, Pyright)
Shared Code	Can have concrete methods	No implementation (signatures only)
Registration	@ABC.register() for virtual subclasses	runtime_checkable decorator
Best For	Internal APIs, frameworks	External code, flexibility

```python
# Using both together
from abc import ABC, abstractmethod
from typing import Protocol, runtime_checkable

@runtime_checkable
class Serializable(Protocol):
    def to_json(self) -> str: ...

# Now you can use isinstance checks
obj = SomeClass()
if isinstance(obj, Serializable):
    print(obj.to_json())
```

Mixins: Reusable Behavior

Small, focused classes for composable functionality

```
class TimestampMixin:
    """Adds timestamp tracking to any class"""
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        from datetime import datetime
        self.created_at = datetime.now()
        self.updated_at = datetime.now()

    def touch(self):
        from datetime import datetime
        self.updated_at = datetime.now()

class SerializableMixin:
    """Adds JSON serialization to any class"""
    def to_dict(self):
        return {k: v for k, v in self.__dict__.items()
                if not k.startswith('_')}

    def to_json(self):
        import json
        return json.dumps(self.to_dict(), default=str)

class User(TimestampMixin, SerializableMixin):
    def __init__(self, name: str, email: str):
        self.name = name
        self.email = email
        super().__init__()  # Initialize mixins

user = User("Alice", "alice@example.com")
print(user.to_json())
# {"name": "Alice", "email": "alice@example.com",
#  "created_at": "2024-01-15 10:30:00", ...}
```

Operator Overloading

Making your classes work with Python operators

> Python uses "magic methods" (dunder methods) to customize operator behavior

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

    def __add__(self, other):
        """v1 + v2"""
        return Vector(self.x + other.x, self.y + other.y)

    def __sub__(self, other):
        """v1 - v2"""
        return Vector(self.x - other.x, self.y - other.y)

    def __mul__(self, scalar):
        """v * 3"""
        return Vector(self.x * scalar, self.y * scalar)

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

v1 = Vector(2, 3)
v2 = Vector(1, 4)
print(v1 + v2)      # Vector(3, 7)
print(v1 * 3)       # Vector(6, 9)
```

Comparison Operators

Making objects comparable and sortable

```python
from functools import total_ordering

@total_ordering  # Generates other comparisons from __eq__ and __lt__
class Student:
    def __init__(self, name, grade):
        self.name = name
        self.grade = grade

    def __eq__(self, other):
        """student1 == student2"""
        return self.grade == other.grade

    def __lt__(self, other):
        """student1 < student2"""
        return self.grade < other.grade

    def __repr__(self):
        return f"{self.name}: {self.grade}"

students = [
    Student("Alice", 85),
    Student("Bob", 92),
    Student("Charlie", 78)
]

print(sorted(students))  # [Charlie: 78, Alice: 85, Bob: 92]
print(max(students))     # Bob: 92
print(students[0] < students[1])  # True
```

Other Useful Operator Methods

CONTAINER OPERATIONS

```
class Inventory:
    def __init__(self):
        self._items = {}

    def __getitem__(self, key):
        """inventory["sword"]"""
        return self._items[key]

    def __setitem__(self, key, value):
        """inventory["sword"] = 5"""
        self._items[key] = value

    def __contains__(self, key):
        """"sword" in inventory"""
        return key in self._items

    def __len__(self):
        """len(inventory)"""
        return len(self._items)

inv = Inventory()
inv["sword"] = 3
print("sword" in inv)  # True
print(len(inv))        # 1
```

CALLABLE OBJECTS

```
class Multiplier:
    def __init__(self, factor):
        self.factor = factor

    def __call__(self, value):
        """Make instance callable"""
        return value * self.factor

double = Multiplier(2)
triple = Multiplier(3)

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

```
# Common use: configurable functions
prices = [10, 20, 30]
print(list(map(double, prices)))
# [20, 40, 60]
```

Functional Programming in Python

Writing cleaner, more expressive code

What is Functional Programming?

> **Key Concepts:**
>
> - **Pure Functions:** Same input → Same output, no side effects
> - **Immutability:** Data doesn't change after creation
> - **First-class Functions:** Functions as values (assign, pass, return)
> - **Higher-order Functions:** Functions that take/return functions

Lambda Functions

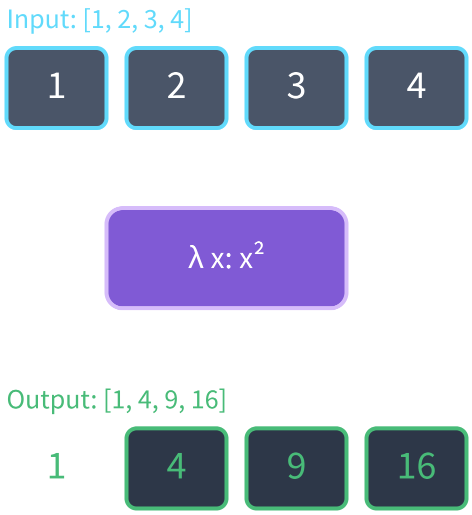
Anonymous, single-expression functions

```python
# Regular function
def square(x):
    return x ** 2

# Lambda equivalent
square = lambda x: x ** 2

# Common use cases
people = [('Alice', 30), ('Bob', 25), ('Charlie', 35)]
sorted_people = sorted(people, key=lambda x: x[1])
# [('Bob', 25), ('Alice', 30), ('Charlie', 35)]

# Multiple arguments
add = lambda x, y: x + y
print(add(5, 3))  # 8

# Conditional expression
max_of_two = lambda a, b: a if a > b else b
print(max_of_two(10, 20))  # 20
```

map() Function

Apply a function to every item in an iterable

```python
# Basic map usage
numbers = [1, 2, 3, 4, 5]
squared = map(lambda x: x**2, numbers)
print(list(squared))
# [1, 4, 9, 16, 25]

# With multiple iterables
list1 = [1, 2, 3]
list2 = [10, 20, 30]
result = map(lambda x, y: x + y, list1, list2)
print(list(result))
# [11, 22, 33]
```



Input: [1, 2, 3, 4]

1 2 3 4

λ x: x^2

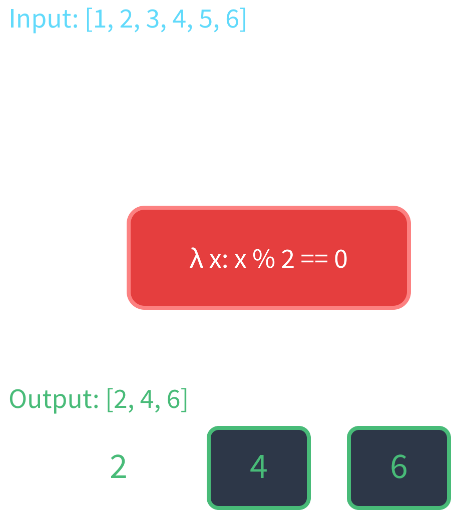
Output: [1, 4, 9, 16]

1 4 9 16

filter() Function

Select items from an iterable based on a condition

```python
# Basic filter usage
numbers = [1, 2, 3, 4, 5, 6]
evens = filter(lambda x: x % 2 == 0, numbers)
print(list(evens))
# [2, 4, 6]

# Filter None/falsy values
data = [0, 1, False, 2, '', 3]
truthy = filter(None, data)
print(list(truthy))
# [1, 2, 3]

# Practical example
users = [
    {'name': 'Alice', 'age': 25},
    {'name': 'Bob', 'age': 17},
    {'name': 'Charlie', 'age': 30}
]
adults = filter(lambda u: u['age'] >= 18, users)
```



Input: [1, 2, 3, 4, 5, 6]

λ x: x % 2 == 0

Output: [2, 4, 6]

reduce() Function

Combine all items in an iterable into a single value

```python
from functools import reduce

# Basic reduce - sum all numbers
numbers = [1, 2, 3, 4, 5]
total = reduce(lambda acc, x: acc + x, numbers)
print(total)  # 15

# With initial value
total = reduce(lambda acc, x: acc + x, numbers, 100)
print(total)  # 115

# Find maximum
maximum = reduce(lambda a, b: a if a > b else b,
numbers)
print(maximum)  # 5

# Practical: flatten nested lists
nested = [[1, 2], [3, 4], [5]]
flat = reduce(lambda acc, lst: acc + lst, nested, [])
# [1, 2, 3, 4, 5]
```

List/Dict/Set Comprehensions

Advanced patterns for data transformation

```
# List comprehension with conditional
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_squares = [x**2 for x in numbers if x % 2 == 0]
# [4, 16, 36, 64, 100]

# Nested list comprehension
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flattened = [num for row in matrix for num in row]
# [1, 2, 3, 4, 5, 6, 7, 8, 9]

# Dict comprehension
words = ['apple', 'banana', 'cherry']
word_lengths = {word: len(word) for word in words}
# {'apple': 5, 'banana': 6, 'cherry': 6}

# Set comprehension - unique values
text = "hello world"
unique_chars = {char.lower() for char in text if
char.isalpha()}
# {'h', 'e', 'l', 'o', 'w', 'r', 'd'}
```

Lazy vs Eager Evaluation

When does Python evaluate expressions?

EAGER (STRICT) EVALUATION

```python
# List comprehension - EAGER
# Computes ALL values immediately
numbers = [x**2 for x in range(1000000)]
# All 1 million values in memory NOW

# Functions evaluate args eagerly
def process(data):
    print("Processing...")
    return data[0]

# This creates the FULL list first
result = process([x**2 for x in range(1000000)])

# Even if we only need the first element!
```

LAZY EVALUATION

```python
# Generator expression - LAZY
# Computes values ON DEMAND
numbers = (x**2 for x in range(1000000))
# Nothing computed yet! Just a recipe

# Only compute what we need
first_five = [next(numbers) for _ in range(5)]
# Only 5 values computed

# Generators are lazy
def lazy_squares(n):
    for i in range(n):
        yield i**2  # Computed when requested

# Itertools - lazy operations
```

```
from itertools import islice
first_10 = list(islice(lazy_squares(1000000), 10))
# Only 10 values ever computed!
```

Lazy Evaluation: Practical Patterns

```python
# Pattern 1: Lazy file processing (don't load entire file)
def process_large_file(filename):
    with open(filename) as f:
        for line in f:  # Lazy - one line at a time
            yield line.strip().upper()

# Pattern 2: Lazy chaining with generators
def numbers():
    n = 0
    while True:  # Infinite!
        yield n
        n += 1

def square(nums):
    for n in nums:
        yield n ** 2

def take(n, iterable):
    for i, item in enumerate(iterable):
        if i >= n:
            break
        yield item

# Lazy pipeline - nothing computed until list()
result = list(take(5, square(numbers())))
# [0, 1, 4, 9, 16] - only 5 values ever computed!

# Pattern 3: Lazy property (computed on first access)
class DataLoader:
    def __init__(self, filename):
        self.filename = filename
        self._data = None  # Not loaded yet

    @property
    def data(self):
        if self._data is None:
            print("Loading data...")  # Only happens once
            self._data = self._load_data()
        return self._data

    def _load_data(self):
        # Expensive operation
        return open(self.filename).read()
```

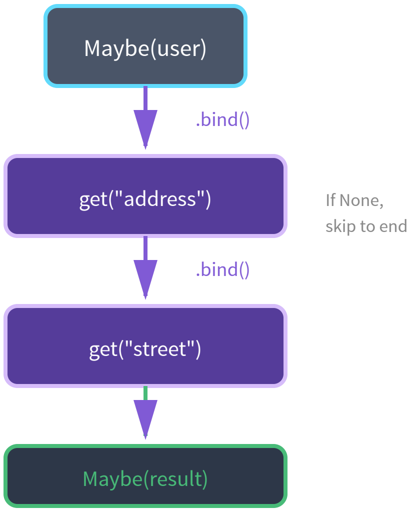
Monads: Chainable Computations

A design pattern for handling values in a context (errors, optionals, async)

THE MAYBE MONAD (OPTIONAL)

```python
# Problem: Chaining operations that might fail
def get_user(id):
    return {"name": "Alice", "address": None}

def get_street(address):
    return address.get("street") if address else None

# Without monad - nested None checks
user = get_user(1)
if user:
    address = user.get("address")
    if address:
        street = address.get("street")
        # ...nightmare!

# With Maybe monad pattern
class Maybe:
    def __init__(self, value):
        self.value = value

    def bind(self, func):
        """Chain operations, skip if None"""
        if self.value is None:
            return Maybe(None)
        return Maybe(func(self.value))

    def __repr__(self):
        return f"Maybe({self.value})"

# Clean chaining!
result = (Maybe(get_user(1))
    .bind(lambda u: u.get("address"))
    .bind(lambda a: a.get("street")))
```

Maybe Monad Flow



Monads: Result Type (Error Handling)

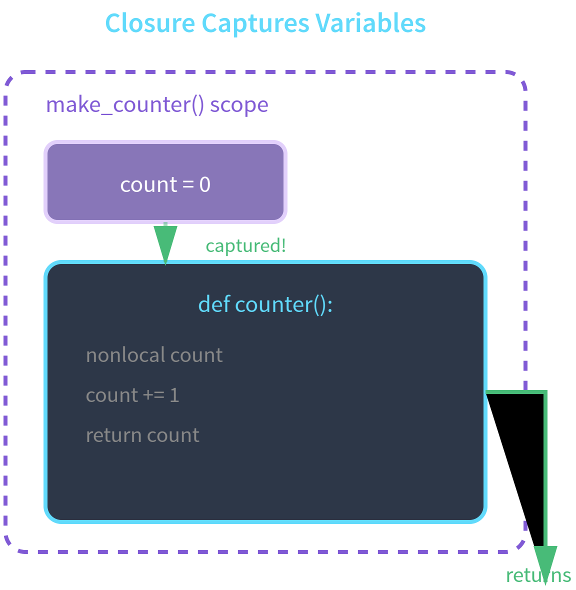
```python
# Result Monad - Either success or failure
class Result:
    def __init__(self, value=None, error=None):
        self.value = value
        self.error = error
        self.is_ok = error is None

    @staticmethod
    def ok(value):
        return Result(value=value)

    @staticmethod
    def err(error):
        return Result(error=error)

    def bind(self, func):
        """Chain operations, propagate errors"""
        if not self.is_ok:
            return self  # Pass through the error
        try:
            return func(self.value)
        except Exception as e:
            return Result.err(str(e))

    def unwrap_or(self, default):
        return self.value if self.is_ok else default

# Usage: Chain operations that might fail
def parse_int(s):
    return Result.ok(int(s))

def divide_by(divisor):
    def divider(n):
        if divisor == 0:
            return Result.err("Division by zero")
        return Result.ok(n / divisor)
    return divider

# Clean error handling chain
result = (Result.ok("42")
    .bind(parse_int)
    .bind(divide_by(2))
    .bind(divide_by(3)))

print(result.value)  # 7.0

# Error propagation
```

```
bad_result = (Result.ok("42")
    .bind(parse_int)
    .bind(divide_by(0))   # Error here
    .bind(divide_by(3)))  # Skipped!

print(bad_result.error)  # "Division by zero"
```

Closures: Functions with Memory

A closure captures variables from its enclosing scope

```python
# A closure is a function that "remembers"
# variables from its enclosing scope

def make_counter():
    count = 0  # This variable is "enclosed"

    def counter():
        nonlocal count  # Access enclosing scope
        count += 1
        return count

    return counter  # Returns the inner function

# Create independent counters
counter_a = make_counter()
counter_b = make_counter()

print(counter_a())  # 1
print(counter_a())  # 2
print(counter_a())  # 3

print(counter_b())  # 1 (separate state!)
print(counter_b())  # 2

# Each closure has its own "memory"
```


Closure Captures Variables
make_counter() scope
count = 0
captured!
def counter():
nonlocal count
count += 1
return count
returns

Closures: Practical Examples

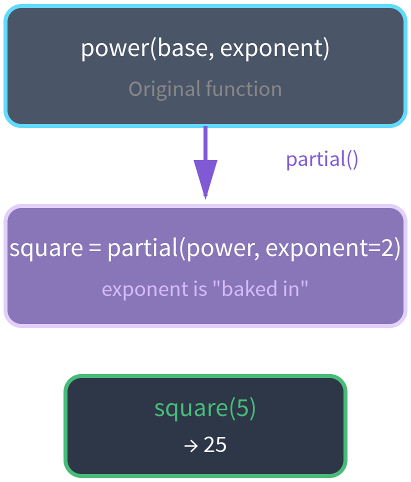
```python
# Example 1: Configurable multiplier
def make_multiplier(factor):
    def multiply(x):
        return x * factor  # 'factor' is captured from enclosing scope
    return multiply

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

# Example 2: Private data (like a simple class)
def create_account(initial_balance):
    balance = initial_balance  # Private variable

    def deposit(amount):
        nonlocal balance
        balance += amount
        return balance

    def withdraw(amount):
        nonlocal balance
        if amount <= balance:
            balance -= amount
            return balance
        return "Insufficient funds"

    def get_balance():
        return balance

    # Return multiple closures sharing the same state
    return deposit, withdraw, get_balance

deposit, withdraw, get_balance = create_account(100)
print(get_balance())  # 100
print(deposit(50))    # 150
print(withdraw(30))   # 120

# Example 3: Event handlers with state
def make_click_handler(button_name):
    clicks = 0
    def handler():
        nonlocal clicks
        clicks += 1
        print(f"{button_name} clicked {clicks} times")
    return handler
```

Partial Functions

Create specialized functions by pre-filling arguments

```
from functools import partial

# Original function with multiple args
def power(base, exponent):
    return base ** exponent

# Create specialized versions
square = partial(power, exponent=2)
cube = partial(power, exponent=3)

print(square(5))  # 25
print(cube(5))    # 125

# Practical: Logging with context
def log(message, level='INFO', prefix=''):
    print(f"[{level}] {prefix}{message}")

error_log = partial(log, level='ERROR')
debug_log = partial(log, level='DEBUG')
api_log = partial(log, prefix='[API] ')

error_log("Connection failed")
# [ERROR] Connection failed
api_log("Request received")
# [INFO] [API] Request received
```

Partial Application



power(base, exponent)

Original function

partial()

square = partial(power, exponent=2)

exponent is "baked in"

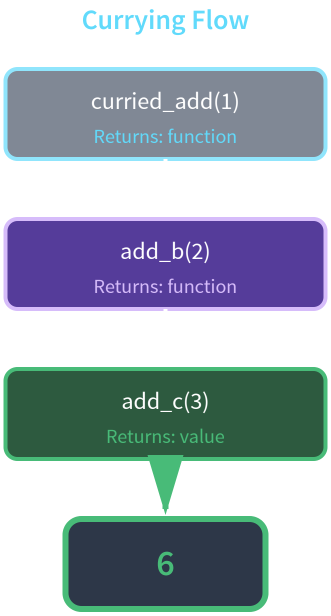
square(5)

→ 25

Currying: Functions Returning Functions

Transform a function with multiple arguments into a chain of single-argument functions

```python
# Regular function
def add(a, b, c):
    return a + b + c

print(add(1, 2, 3))  # 6

# Curried version - each call returns a new function
def curried_add(a):
    def add_b(b):
        def add_c(c):
            return a + b + c
        return add_c
    return add_b

# Call one argument at a time
print(curried_add(1)(2)(3))  # 6

# Store intermediate functions
add_1 = curried_add(1)
add_1_2 = add_1(2)
result = add_1_2(3)  # 6

# Practical: configurable formatter
def make_formatter(prefix):
    def with_suffix(suffix):
        def format_value(value):
            return f"{prefix}{value}{suffix}"
        return format_value
    return with_suffix

price_fmt = make_formatter("$")("USD")
print(price_fmt(99.99))  # $99.99USD
```


Currying Flow
curried_add(1)
Returns: function
add_b(2)
Returns: function
add_c(3)
Returns: value
6

Currying vs Partial: When to Use Which

CURRYING

```python
# Manual currying
def multiply(a):
    def by(b):
        return a * b
    return by

double = multiply(2)
triple = multiply(3)

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

# Auto-curry helper
def curry(func):
    def curried(*args):
        if len(args) >= func.__code__.co_argcount:
            return func(*args)
        return lambda *more: curried(*(args + more))
    return curried

@curry
def add3(a, b, c):
    return a + b + c

print(add3(1)(2)(3))    # 6
print(add3(1, 2)(3))    # 6
print(add3(1)(2, 3))    # 6
```

PARTIAL APPLICATION

```python
from functools import partial

# Partial is more Pythonic
def greet(greeting, name, punctuation):
    return f"{greeting}, {name}{punctuation}"

# Fix some arguments
say_hello = partial(greet, "Hello")
excited_hello = partial(greet, "Hello",
punctuation="!")
```

```
print(say_hello("Alice", "!"))  # Hello, Alice!
print(excited_hello("Bob"))     # Hello, Bob!
```

USE CURRYING WHEN:

- Building DSLs or fluent APIs
- Functional programming patterns

USE PARTIAL WHEN:

- Creating specialized versions of functions
- Callbacks with pre-set arguments
- More readable, Pythonic code

functools: lru_cache for Memoization

Cache expensive computations automatically

```python
from functools import lru_cache

# Without caching - exponential time O(2^n)
def fib_slow(n):
    if n < 2:
        return n
    return fib_slow(n-1) + fib_slow(n-2)

# With caching - linear time O(n)
@lru_cache(maxsize=128)
def fib_fast(n):
    if n < 2:
        return n
    return fib_fast(n-1) + fib_fast(n-2)

print(fib_fast(100))  # Instant: 354224848179261915075
print(fib_fast.cache_info())  # CacheInfo(hits=98, misses=101, ...)

# Practical: API response caching
@lru_cache(maxsize=100)
def fetch_user(user_id):
    """Expensive API call - cached!"""
    return api.get(f'/users/{user_id}')

# Clear cache when needed
fib_fast.cache_clear()
```

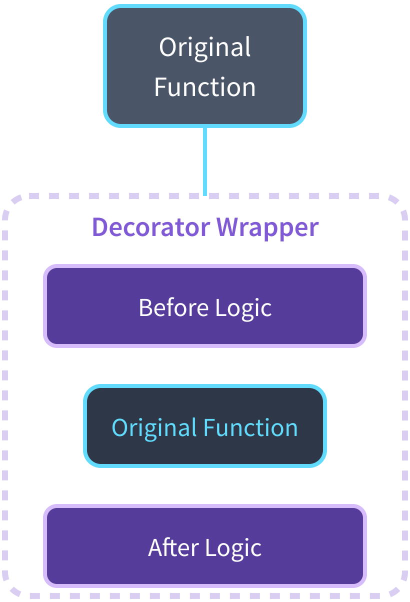
Decorators & Generators

Advanced Python patterns for cleaner, more efficient code

What are Decorators?

Functions that modify the behavior of other functions

```python
# Decorator syntax
@decorator_name
def my_function():
    pass

# Is equivalent to:
def my_function():
    pass
my_function = decorator_name(my_function)

# Simple example
def uppercase_decorator(func):
    def wrapper():
        result = func()
        return result.upper()
    return wrapper

@uppercase_decorator
def greet():
    return "hello world"

print(greet())  # "HELLO WORLD"
```


Original Function
Decorator Wrapper
Before Logic
Original Function
After Logic

Creating Basic Decorators

Essential patterns and best practices

```python
from functools import wraps
import time

# Timer decorator
def timer(func):
    @wraps(func)  # Preserves original function's metadata
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"{func.__name__} took {end - start:.4f} seconds")
        return result
    return wrapper

@timer
def slow_function():
    time.sleep(1)
    return "Done"

# Logging decorator
def log_calls(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__} with args={args}")
        result = func(*args, **kwargs)
        print(f"{func.__name__} returned {result}")
        return result
    return wrapper

@log_calls
def add(a, b):
    return a + b
```

Decorators with Arguments

Creating configurable decorators

```
# Decorator factory pattern
def repeat(times):
    """Decorator that repeats function execution"""
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            results = []
            for _ in range(times):
                result = func(*args, **kwargs)
                results.append(result)
            return results
        return wrapper
    return decorator

@repeat(times=3)
def greet(name):
    return f"Hello, {name}!"

print(greet("Alice"))
# ['Hello, Alice!', 'Hello, Alice!', 'Hello, Alice!']

# Retry decorator
def retry(max_attempts=3, delay=1):
    """Retry function on exception"""
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            for attempt in range(max_attempts):
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    if attempt == max_attempts - 1:
                        raise
                    time.sleep(delay)
        return wrapper
    return decorator
```

Common Built-in Decorators

@property, @staticmethod, @classmethod

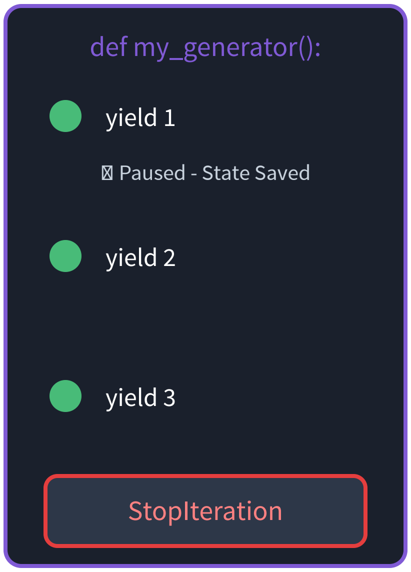
```python
class Person:
    def __init__(self, first_name, birth_year):
        self._first_name = first_name
        self._birth_year = birth_year

    @property
    def age(self):
        from datetime import datetime
        return datetime.now().year - self._birth_year

    @property
    def first_name(self):
        return self._first_name

    @first_name.setter
    def first_name(self, value):
        if not value:
            raise ValueError("Name cannot be empty")
        self._first_name = value.capitalize()

    @staticmethod
    def is_adult(age):
        return age >= 18

    @classmethod
    def from_birth_year(cls, first_name, birth_year):
        return cls(first_name, birth_year)

person = Person("john", 1990)
print(person.age)         # Computed property
print(Person.is_adult(20))  # Static method
```

What are Generators?

Functions that produce values lazily using yield

```python
# Regular function - returns all at once
def get_numbers_list(n):
    result = []
    for i in range(n):
        result.append(i ** 2)
    return result

# Generator - yields one at a time
def get_numbers_generator(n):
    for i in range(n):
        yield i ** 2

# Usage
gen = get_numbers_generator(5)
print(next(gen))  # 0
print(next(gen))  # 1
print(next(gen))  # 4

# Infinite generator
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

fib = fibonacci()
first_10 = [next(fib) for _ in range(10)]
```

Generator Flow



def my_generator():

● yield 1

⏸ Paused - State Saved

● yield 2

● yield 3

StopIteration

Generator Expressions vs List Comprehensions

Choosing the right tool for memory efficiency

```python
# List comprehension - Creates entire list in memory
squares = [x**2 for x in range(1000000)]
# Memory: ~8MB for 1 million integers

# Generator expression - Creates generator object
squares_gen = (x**2 for x in range(1000000))
# Memory: ~200 bytes!

import sys
print(sys.getsizeof([x**2 for x in range(1000000)]))
# 8448728 bytes
print(sys.getsizeof((x**2 for x in range(1000000))))
# 200 bytes

# Use List Comprehension when:
# - Need to iterate multiple times
# - Need indexing/slicing
# - Dataset is small

# Use Generator Expression when:
# - Iterate once
# - Large or infinite sequences
# - Memory is a concern
```

itertools Module Highlights

Powerful tools for working with iterators

```python
from itertools import count, cycle, chain, islice, groupby, combinations, permutations

# count() - infinite counter
for i in count(10, 2):
    if i > 20: break
    print(i)  # 10, 12, 14, 16, 18, 20

# chain() - combine iterables
combined = list(chain([1, 2, 3], [4, 5, 6]))  # [1, 2, 3, 4, 5, 6]

# islice() - slice an iterator
first_5 = list(islice(count(), 5))  # [0, 1, 2, 3, 4]

# combinations() and permutations()
items = ['A', 'B', 'C']
print(list(combinations(items, 2)))  # [('A', 'B'), ('A', 'C'), ('B', 'C')]
print(list(permutations(items, 2)))  # All orderings of 2 items

# accumulate() - running totals
from itertools import accumulate
numbers = [1, 2, 3, 4, 5]
print(list(accumulate(numbers)))  # [1, 3, 6, 10, 15]
```

Generator Pipelines

Chain generators to create efficient data processing pipelines

```python
def read_lines(filename):
    """Stage 1: Read lines from file"""
    with open(filename) as f:
        for line in f:
            yield line.strip()

def filter_comments(lines):
    """Stage 2: Remove comment lines"""
    for line in lines:
        if not line.startswith('#'):
            yield line

def parse_csv(lines):
    """Stage 3: Parse CSV fields"""
    for line in lines:
        yield line.split(',')

def extract_field(records, index):
    """Stage 4: Extract specific field"""
    for record in records:
        if len(record) > index:
            yield record[index]

# Build the pipeline - no data processed yet!
pipeline = extract_field(
    parse_csv(
        filter_comments(
            read_lines('data.csv')
        )
    ),
    index=2
)

# Process lazily - one line at a time through entire
pipeline
for value in pipeline:
    print(value)
```

Why Generator Pipelines?

MEMORY EFFICIENT

```
# Without generators - loads entire file
data = open('huge.log').readlines()  # 10GB!
filtered = [l for l in data if 'ERROR' in l]
results = [parse(l) for l in filtered]

# With generator pipeline
def find_errors(filename):
    for line in open(filename):
        if 'ERROR' in line:
            yield parse(line)

# Only one line in memory at a time!
for error in find_errors('huge.log'):
    process(error)
```

COMPOSABLE & REUSABLE

```
# Reusable pipeline stages
def uppercase(items):
    for item in items:
        yield item.upper()

def add_prefix(items, prefix):
    for item in items:
        yield f"{prefix}{item}"

# Compose pipelines
words = ['hello', 'world']
result = add_prefix(
    uppercase(words),
    prefix='>>> '
)

print(list(result))
# ['>>> HELLO', '>>> WORLD']
```

Coroutines with send()

Push-based data flow: send values INTO generators

```
def averager():
    """Coroutine that computes running average"""
    total = 0.0
    count = 0
    average = None
    while True:
        value = yield average  # Receive value, send back average
        total += value
        count += 1
        average = total / count

# Create and prime the coroutine
avg = averager()
next(avg)  # Prime: advance to first yield

# Send values in and get running average back
print(avg.send(10))  # 10.0
print(avg.send(20))  # 15.0
print(avg.send(30))  # 20.0
print(avg.send(40))  # 25.0

# The coroutine maintains state between sends!
```

Coroutine Patterns

PRIMING DECORATOR

```python
def coroutine(func):
    """Auto-prime coroutines"""
    def wrapper(*args, **kwargs):
        gen = func(*args, **kwargs)
        next(gen)  # Prime it
        return gen
    return wrapper

@coroutine
def grep(pattern):
    """Filter lines by pattern"""
    while True:
        line = yield
        if pattern in line:
            print(f"Found: {line}")

# No need to call next()!
searcher = grep("ERROR")
searcher.send("INFO: All good")
searcher.send("ERROR: Disk full")
# Found: ERROR: Disk full
```

PIPELINE SINK

```python
@coroutine
def writer(filename):
    """Coroutine sink - writes to file"""
    with open(filename, 'w') as f:
        while True:
            line = yield
            f.write(line + '\n')
            f.flush()

# Use in data pipeline
output = writer('results.txt')
for line in data_source():
    if should_save(line):
        output.send(line)
```

```
# Close when done
output.close()
```

itertools: Python's Stream Processing

Functional tools equivalent to Java Streams

.filter()	filter(), filterfalse()	Select elements
.map()	map(), starmap()	Transform elements
.flatMap()	chain.from_iterable()	Flatten nested
.limit()	islice()	Take first N
.skip()	islice(iter, n, None)	Skip first N
.distinct()	dict.fromkeys() or set	Remove duplicates
.sorted()	sorted()	Sort elements
.reduce()	functools.reduce()	Aggregate to single value
.collect()	list(), set(), dict()	Terminal operation
.groupingBy()	groupby()	Group by key

Stream-Style Processing with itertools

```python
from itertools import groupby, starmap, chain, takewhile, dropwhile
from functools import reduce
from operator import mul

data = [
    {'name': 'Alice', 'dept': 'Engineering', 'salary': 95000},
    {'name': 'Bob', 'dept': 'Sales', 'salary': 75000},
    {'name': 'Carol', 'dept': 'Engineering', 'salary': 110000},
    {'name': 'Dave', 'dept': 'Sales', 'salary': 82000},
]

# Java: stream().filter().map().collect()
# Python equivalent:
engineers = [e['name'] for e in data if e['dept'] == 'Engineering']

# Using functional style with itertools
high_earners = list(filter(lambda e: e['salary'] > 80000, data))
names = list(map(lambda e: e['name'], high_earners))

# Group by department (like Collectors.groupingBy)
sorted_data = sorted(data, key=lambda x: x['dept'])
for dept, group in groupby(sorted_data, key=lambda x: x['dept']):
    members = list(group)
    total = sum(e['salary'] for e in members)
    print(f"{dept}: ${total:,} ({len(members)} people)")
# Engineering: $205,000 (2 people)
# Sales: $157,000 (2 people)
```

Advanced itertools Patterns

```python
from itertools import (
    tee, pairwise, batched,
    takewhile, dropwhile,
    zip_longest, product
)

# pairwise (Python 3.10+)
nums = [1, 2, 3, 4, 5]
for a, b in pairwise(nums):
    print(f"{a} -> {b}")
# 1 -> 2, 2 -> 3, 3 -> 4, 4 -> 5

# batched (Python 3.12+)
items = range(10)
for batch in batched(items, 3):
    print(batch)
# (0, 1, 2), (3, 4, 5), (6, 7, 8), (9,)

# takewhile / dropwhile
data = [1, 3, 5, 2, 4, 6]
print(list(takewhile(lambda x: x < 4, data)))
# [1, 3]  # Stops at first False
```

```python
# Cartesian product
colors = ['red', 'blue']
sizes = ['S', 'M', 'L']
for combo in product(colors, sizes):
    print(combo)
# ('red', 'S'), ('red', 'M'), ...

# tee - duplicate an iterator
original = iter([1, 2, 3])
copy1, copy2 = tee(original, 2)
print(list(copy1))  # [1, 2, 3]
print(list(copy2))  # [1, 2, 3]

# zip_longest for uneven iterables
a = [1, 2, 3]
b = ['a', 'b']
```

```
result = list(zip_longest(a, b, fillvalue='-'))
# [(1, 'a'), (2, 'b'), (3, '-')]
```

Building a Data Pipeline with itertools

```python
from itertools import chain, islice, filterfalse
from functools import reduce

# Simulating Java Streams API fluency in Python
class Stream:
    def __init__(self, iterable):
        self.data = iter(iterable)

    def filter(self, predicate):
        self.data = filter(predicate, self.data)
        return self

    def map(self, func):
        self.data = map(func, self.data)
        return self

    def limit(self, n):
        self.data = islice(self.data, n)
        return self

    def collect(self):
        return list(self.data)

    def reduce(self, func, initial=None):
        return reduce(func, self.data, initial) if
initial else reduce(func, self.data)

# Usage - fluent API like Java Streams!
result = (Stream(range(100))
    .filter(lambda x: x % 2 == 0)
    .map(lambda x: x ** 2)
    .limit(5)
    .collect())

print(result)  # [0, 4, 16, 36, 64]
```

Data Handling

Working with External Data Formats

- JSON - JavaScript Object Notation
- CSV - Comma Separated Values
- Regular Expressions - Pattern Matching
- DateTime and File Handling

Working with JSON

PYTHON TO JSON (ENCODING)

```python
import json

data = {
    "name": "Alice",
    "age": 30,
    "skills": ["Python", "Java"]
}

# Convert to JSON string
json_string = json.dumps(data, indent=2)

# Write to file
with open('data.json', 'w') as f:
    json.dump(data, f, indent=2)
```

JSON TO PYTHON (DECODING)

```python
import json

# JSON string to Python
json_string = '{"name": "Bob", "age": 25}'
data = json.loads(json_string)
print(data['name'])  # Bob

# Read from file
with open('data.json', 'r') as f:
    data = json.load(f)
```

Working with CSV Files

READING WITH CSV.READER()

```python
import csv

with open('data.csv', 'r') as file:
    csv_reader = csv.reader(file)
    header = next(csv_reader)

    for row in csv_reader:
        name, age, city = row
        print(f"{name} is {age}")
```

WRITING WITH CSV.WRITER()

```python
import csv

with open('output.csv', 'w', newline='') as file:
    csv_writer = csv.writer(file)
    csv_writer.writerow(['Name', 'Age', 'City'])
    csv_writer.writerow(['Alice', 30, 'NYC'])
    csv_writer.writerows([
        ['Bob', 25, 'LA'],
        ['Carol', 28, 'Chicago']
    ])
```

Regular Expressions (Regex)

PATTERN MATCHING POWER

- Powerful text pattern matching
- Search, validate, extract data
- Format validation (email, phone)

```
Text: "Email: john@example.com"

Pattern: \w+@\w+\.\w+
```

```
john@example.com
```

The re Module Functions

```python
import re

text = "Contact: alice@example.com and bob@test.org"

# search() - finds ANYWHERE in string
result = re.search(r'\w+@\w+\.\w+', text)
print(result.group())  # alice@example.com

# findall() - returns list of all matches
emails = re.findall(r'\w+@\w+\.\w+', text)
print(emails)  # ['alice@example.com', 'bob@test.org']

# sub() - find and replace
masked = re.sub(r'\w+@\w+\.\w+', '[EMAIL]', text)
print(masked)  # Contact: [EMAIL] and [EMAIL]

# Capture groups
pattern = r'(\w+)@(\w+)\.(\w+)'
match = re.search(pattern, text)
print(match.group(1))  # alice (username)
```

Working with datetime Module

```
from datetime import datetime, date, timedelta

# Current date and time
now = datetime.now()
today = date.today()

# Date arithmetic
future = now + timedelta(days=7)
past = now - timedelta(hours=5)

# Time difference
date1 = datetime(2024, 1, 1)
date2 = datetime(2024, 12, 31)
diff = date2 - date1
print(f"Days between: {diff.days}")

# Formatting (datetime → string)
print(now.strftime("%Y-%m-%d"))       # 2024-01-15
print(now.strftime("%B %d, %Y"))      # January 15, 2024

# Parsing (string → datetime)
date_str = "2024-01-15"
dt = datetime.strptime(date_str, "%Y-%m-%d")
```

Modern File Handling with pathlib

```python
from pathlib import Path

# Create path objects
path = Path('data/files/report.txt')

# Path properties
print(path.name)      # report.txt
print(path.stem)      # report
print(path.suffix)    # .txt
print(path.parent)    # data/files

# Read and write
path = Path('data.txt')
content = path.read_text()
path.write_text('New content')

# Join paths (cross-platform)
file_path = Path('data') / 'reports' / 'jan.txt'

# Create directories
Path('data/new_folder').mkdir(parents=True,
exist_ok=True)

# List files with glob
for file in Path('data').glob('*.txt'):
    print(file)
```

Testing in Python

Writing Reliable, Maintainable Code

unittest Module Basics

CODE TO TEST

```python
# calculator.py
def add(a, b):
    return a + b

def multiply(a, b):
    return a * b
```

TEST FILE

```python
import unittest
from calculator import add, multiply

class TestCalculator(unittest.TestCase):
    def test_add(self):
        self.assertEqual(add(2, 3), 5)
        self.assertEqual(add(-1, 1), 0)

    def test_multiply(self):
        self.assertEqual(multiply(3, 4), 12)

if __name__ == '__main__':
    unittest.main()
```

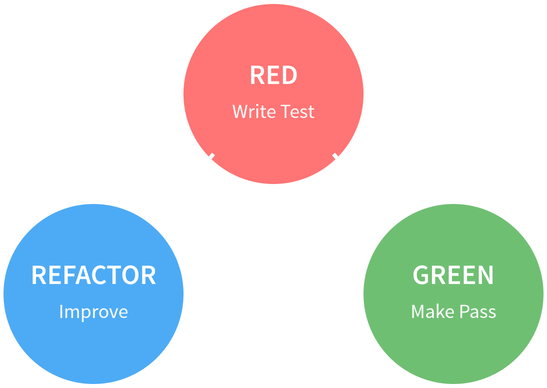
pytest: Modern Testing

```
# pytest style (simpler!)
def test_addition():
    assert 2 + 2 == 4

def test_subtraction():
    assert 5 - 3 == 2
```

PYTEST FEATURES

- Simple assertions
- Better output
- Fixtures
- Parametrize

Test-Driven Development (TDD)



TDD WORKFLOW

1. RED - Write failing test
2. GREEN - Make it pass
3. REFACTOR - Clean up
4. **Repeat**

Python Packaging

Organizing and Distributing Your Code

Virtual Environments (venv)

WHY VIRTUAL ENVIRONMENTS?

- **Isolation** - Each project has own packages
- **No conflicts** - Different versions
- **Reproducible** - Share dependencies

```
# Create virtual environment
python -m venv venv

# Activate (Linux/Mac)
source venv/bin/activate

# Activate (Windows)
venv\Scripts\activate

# Install packages
pip install requests pandas

# Deactivate
deactivate
```

Project Structure

```
myproject/
├── myproject/          # Package directory
│   ├── __init__.py     # Makes it a package
│   ├── main.py         # Main module
│   └── utils.py        # Utility functions
├── tests/              # Test directory
│   ├── __init__.py
│   └── test_main.py
├── venv/               # Virtual environment
├── .gitignore          # Git ignore file
├── requirements.txt    # Dependencies
├── pyproject.toml      # Package metadata
└── README.md           # Project readme
```

The collections Module

Specialized container data types beyond lists and dicts

> ### KEY TYPES
>
> - **Counter** - Count hashable objects
> - **defaultdict** - Dict with default factory
> - **deque** - Double-ended queue
> - **namedtuple** - Tuple with named fields
> - **OrderedDict** - Remembers insertion order
> - **ChainMap** - Combine multiple dicts

Counter: Counting Made Easy

```python
from collections import Counter

# Count elements in a list
colors = ['red', 'blue', 'red', 'green', 'blue', 'blue']
color_count = Counter(colors)
print(color_count)  # Counter({'blue': 3, 'red': 2, 'green': 1})

# Count characters in a string
text = "mississippi"
char_count = Counter(text)
print(char_count)  # Counter({'i': 4, 's': 4, 'p': 2, 'm': 1})

# Most common elements
print(char_count.most_common(2))  # [('i', 4), ('s', 4)]

# Arithmetic with Counters
inventory1 = Counter(apples=5, oranges=3)
inventory2 = Counter(apples=2, bananas=4)

print(inventory1 + inventory2)  # Counter({'apples': 7, 'bananas': 4, 'oranges': 3})
print(inventory1 - inventory2)  # Counter({'oranges': 3, 'apples': 3})

# Count words in text
words = "the quick brown fox jumps over the lazy dog".split()
word_freq = Counter(words)
print(word_freq.most_common(3))  # [('the', 2), ('quick', 1), ('brown', 1)]
```

defaultdict: Automatic Default Values

```python
from collections import defaultdict

# Without defaultdict - KeyError or verbose code
regular_dict = {}
# regular_dict['missing']  # KeyError!
regular_dict.setdefault('colors', []).append('red')  # 
Verbose!

# With defaultdict - clean and simple
dd_list = defaultdict(list)
dd_list['colors'].append('red')
dd_list['colors'].append('blue')
print(dd_list)  # defaultdict(<class 'list'="">,
{'colors': ['red', 'blue']})

# Group items by category
products = [
    ('fruit', 'apple'), ('vegetable', 'carrot'),
    ('fruit', 'banana'), ('vegetable', 'lettuce')
]
grouped = defaultdict(list)
for category, item in products:
    grouped[category].append(item)
print(dict(grouped))  # {'fruit': ['apple', 'banana'],
'vegetable': ['carrot', 'lettuce']}

# Count with defaultdict(int)
word_count = defaultdict(int)
for word in "the quick brown fox".split():
    word_count[word] += 1
print(dict(word_count))  # {'the': 1, 'quick': 1,
'brown': 1, 'fox': 1}
        </class>
```

deque: Fast Double-Ended Queue

```python
from collections import deque

# Create a deque
d = deque(['a', 'b', 'c'])

# O(1) operations on both ends
d.append('d')          # Right: ['a','b','c','d']
d.appendleft('z')      # Left: ['z','a','b','c','d']
d.pop()                # Remove right: 'd'
d.popleft()            # Remove left: 'z'

# Rotate elements
d = deque([1, 2, 3, 4, 5])
d.rotate(2)    # [4, 5, 1, 2, 3]
d.rotate(-2)   # [1, 2, 3, 4, 5]

# Fixed-size buffer
buffer = deque(maxlen=3)
buffer.append(1)  # [1]
buffer.append(2)  # [1, 2]
buffer.append(3)  # [1, 2, 3]
buffer.append(4)  # [2, 3, 4] - 1 removed!
```

PERFORMANCE COMPARISON

append right	O(1)	O(1)
pop right	O(1)	O(1)
append left	O(n)	O(1)
pop left	O(n)	O(1)

> - Recent items buffer

- Undo/redo stacks
- BFS traversal queue
- Sliding window

Other Useful Collections

NAMEDTUPLE

```
from collections import namedtuple

# Create a named tuple type
Point = namedtuple('Point', ['x', 'y'])
Color = namedtuple('Color', 'r g b')

# Use like a class but immutable
p = Point(3, 4)
print(p.x, p.y)        # 3 4
print(p[0], p[1])      # 3 4

# Convert to dict
print(p._asdict())
# {'x': 3, 'y': 4}

# Create new with replaced value
p2 = p._replace(x=10)
print(p2)  # Point(x=10, y=4)
```

CHAINMAP

```
from collections import ChainMap

# Combine dicts (first match wins)
defaults = {'color': 'red', 'size': 'medium'}
user_prefs = {'color': 'blue'}
runtime = {'debug': True}

config = ChainMap(runtime, user_prefs, defaults)
print(config['color'])  # 'blue'
print(config['size'])   # 'medium'
print(config['debug'])  # True

# Great for configuration layers
import os
env_vars = os.environ
config = ChainMap(env_vars, defaults)
```

The os and sys Modules

Interacting with the operating system and Python runtime

> - **os** - Operating system interface: files, directories, environment
> - **sys** - Python interpreter: arguments, paths, version info

os: File System Operations

```
import os

# Current working directory
print(os.getcwd())           # /home/user/project
os.chdir('/tmp')             # Change directory

# List directory contents
files = os.listdir('.')      # ['file1.txt', 'folder']
files = os.listdir('/home')  # List specific directory

# Create and remove directories
os.mkdir('new_folder')       # Create single directory
os.makedirs('a/b/c')         # Create nested
directories
os.rmdir('new_folder')       # Remove empty directory
os.removedirs('a/b/c')       # Remove nested empty
directories

# File operations
os.rename('old.txt', 'new.txt')
os.remove('file.txt')        # Delete file

# Check paths
os.path.exists('/tmp')       # True
os.path.isfile('file.txt')   # True if file
os.path.isdir('/tmp')        # True if directory
os.path.getsize('file.txt')  # Size in bytes
```

os.path: Path Manipulation

```
import os
from pathlib import Path  # Modern alternative

# Classic os.path operations
path = '/home/user/documents/report.pdf'

os.path.basename(path)    # 'report.pdf'
os.path.dirname(path)     # '/home/user/documents'
os.path.split(path)       # ('/home/user/documents',
'report.pdf')
os.path.splitext(path)    #
('/home/user/documents/report', '.pdf')

# Build paths safely (handles OS-specific separators)
os.path.join('home', 'user', 'file.txt')  #
'home/user/file.txt'

# Expand user and environment variables
os.path.expanduser('~/docs')    # '/home/user/docs'
os.path.expandvars('$HOME/docs')  # '/home/user/docs'

# Modern pathlib alternative (Python 3.4+)
path = Path('/home/user/documents/report.pdf')
print(path.name)      # 'report.pdf'
print(path.stem)      # 'report'
print(path.suffix)    # '.pdf'
print(path.parent)    # Path('/home/user/documents')
print(path.exists())  # True/False

# Iterate over directory
for f in Path('.').glob('*.py'):
    print(f)
```

os: Environment Variables

```python
import os

# Get environment variables
home = os.environ.get('HOME')
path = os.environ.get('PATH')
api_key = os.environ.get('API_KEY', 'default')

# All environment variables
for key, value in os.environ.items():
    if key.startswith('PYTHON'):
        print(f"{key}={value}")

# Set environment variable
os.environ['MY_VAR'] = 'my_value'

# Process information
print(os.getpid())     # Process ID
print(os.getppid())    # Parent process ID

# Platform info
print(os.name)         # 'posix', 'nt', 'java'
print(os.sep)          # '/' or '\\'
print(os.linesep)      # '\n' or '\r\n'
```

WALKING DIRECTORY TREES

```python
import os

# Walk through all directories
for root, dirs, files in os.walk('/project'):
    # Skip hidden and venv directories
    dirs[:] = [d for d in dirs
               if not d.startswith('.')
               and d != 'venv']

    for file in files:
        if file.endswith('.py'):
            full_path = os.path.join(root, file)
            print(full_path)

# Find all Python files recursively
```

```
from pathlib import Path
py_files = list(Path('.').rglob('*.py'))
```

sys: Python Runtime Information

```python
import sys

# Command line arguments
# python script.py arg1 arg2
print(sys.argv)       # ['script.py', 'arg1', 'arg2']
print(sys.argv[0])    # Script name
print(sys.argv[1:])   # Arguments only

# Python version
print(sys.version)           # '3.11.0 (main, Oct 24 2022, ...)'
print(sys.version_info)      # sys.version_info(major=3, minor=11, micro=0, ...)
print(sys.version_info.major)  # 3

# Module search path (where Python looks for imports)
print(sys.path)       # List of directories
sys.path.append('/my/custom/modules')  # Add custom path

# Platform information
print(sys.platform)   # 'linux', 'darwin', 'win32'

# Standard I/O streams
sys.stdout.write("Hello\n")  # Same as print()
sys.stderr.write("Error\n")  # Write to stderr
```

sys: Memory and Exit Control

```python
import sys

# Exit program
sys.exit(0)      # Success
sys.exit(1)      # Error
sys.exit("Error message")

# Memory usage
obj = [1, 2, 3, 4, 5]
print(sys.getsizeof(obj))    # 104 bytes

# Get size of nested objects
def total_size(obj):
    size = sys.getsizeof(obj)
    if isinstance(obj, dict):
        size += sum(total_size(v)
                    for v in obj.values())
    elif hasattr(obj, '__iter__'):
        size += sum(total_size(i)
                    for i in obj)
    return size

# Reference count
x = [1, 2, 3]
print(sys.getrefcount(x))  # 2+
```

COMMON PATTERNS

```python
import sys
import os

# Cross-platform script
if sys.platform == 'win32':
    config_dir = os.path.expandvars('%APPDATA%')
else:
    config_dir = os.path.expanduser('~/.config')

# Version check
if sys.version_info < (3, 8):
    sys.exit("Python 3.8+ required")

# Command-line tool
def main():
    if len(sys.argv) < 2:
        print("Usage: script.py <file>")
        sys.exit(1)

    filename = sys.argv[1]
    # ... process file

if __name__ == '__main__':
    main()
                </file>
```

Course Summary

OOP

- Classes & Objects
- Inheritance
- ABC & Protocol
- Mixins
- Operator Overloading

FUNCTIONAL

- Lambda functions
- map/filter/reduce
- Decorators
- Generators & Pipelines
- Coroutines

STANDARD LIBRARY

- itertools
- collections
- os & sys
- pathlib

PRO

- Testi
- Data
- Pack
- Best

Thank You!

Keep Coding, Keep Learning

"The only way to learn a new programming language is by writing programs in it."
- Dennis Ritchie

Speaker notes