Lecture 9 Handout

Advanced Object-Oriented Programming: Inheritance and Polymorphism

INF 605 - Introduction to Programming - Python

Prof. Rongyu Lin
Quinnipiac University
School of Computing and Engineering

Fall 2025

Required Reading

Textbook: Chapter 10, Sections 10.7-10.11 (Inheritance, Polymorphism, and Advanced OOP)
Reference Notebooks: `ch10/10_07.ipynb` (Inheritance), `ch10/10_08.ipynb` (Building Hierarchies), `ch10/10_09.ipynb` (Duck Typing), `ch10/10_10.*.ipynb` (Operator Overloading), `ch10/10_11.ipynb` (Exception Hierarchies)

Learning Objectives

By the end of this lecture, you will be able to:

1. **Master inheritance concepts** creating base classes and derived classes with proper syntax
2. **Implement method overriding** to specialize behavior in derived classes
3. **Understand polymorphism** enabling flexible, extensible code design
4. **Apply duck typing principles** for dynamic polymorphic behavior
5. **Create class hierarchies** modeling real-world relationships effectively
6. **Implement operator overloading** using special methods like __add__, __eq__, __lt__
7. **Design custom exception hierarchies** extending built-in exception classes
8. **Understand super() function** for calling parent class methods properly
9. **Master string fundamentals** including creation, indexing, slicing, methods, formatting, and immutability
10. **Compare inheritance vs composition** choosing appropriate design patterns

Prerequisites Review

Building on Your Advanced OOP Foundation:

From Lecture 8: Object-oriented programming foundation - class creation mastery, object instantiation, __init__ method expertise, instance methods and attributes, string representations using __str__ and __repr__, properties and encapsulation using @property decorator, single class design following professional principles

Earlier Foundations: Complete programming arsenal including advanced data structures, exception handling expertise, function architecture mastery, sophisticated data processing, module system utilization, complete data type fluency, algorithmic thinking excellence

Transformation Goal: Evolve from **single class design** to **class hierarchies and polymorphic design** - creating sophisticated multi-class systems with inheritance relationships and polymorphic behavior.

1 Part 1: Understanding Inheritance

1.1 The Inheritance Concept

Inheritance allows a class to inherit attributes and methods from another class, creating an "is-a" relationship where the derived class is a specialized version of the base class. This fundamental concept enables code reuse and creates natural hierarchical relationships that mirror real-world classifications.

Think of inheritance like family relationships where children inherit characteristics from their parents while also developing their own unique traits. A child inherits their parent's last name, family traditions, and genetic characteristics, but also develops their own personality, skills, and interests. Similarly, in programming, a derived class inherits all the capabilities of its parent class while adding or modifying functionality specific to its purpose.

Consider how biological taxonomy works: all mammals share certain characteristics (warm-blooded, have hair, nurse their young), but specific mammals like dogs and cats have their own specialized behaviors and traits while still maintaining their mammalian characteristics.

```python
# Basic inheritance syntax demonstration
# Base class (parent) defines common characteristics
class Animal:
    def __init__(self, name, species):
        self.name = name           # Common to all animals
        self.species = species     # Common to all animals

    def make_sound(self):
        return "Some generic animal sound"  # Basic behavior

    def sleep(self):
        return f"{self.name} is sleeping"   # Shared behavior

# Derived class (child) inherits from Animal
class Dog(Animal):  # Dog inherits everything from Animal
    def __init__(self, name, breed):
        super().__init__(name, "Canine")  # Call parent constructor
        self.breed = breed                # Add dog-specific attribute

    def make_sound(self):  # Override parent method
        return "Woof! Woof!"  # Specialized behavior for dogs

```

```python
# Creating objects to demonstrate inheritance
animal = Animal("Generic", "Unknown")
dog = Dog("Buddy", "Golden Retriever")

print(f"Animal sound: {animal.make_sound()}")
print(f"Dog sound: {dog.make_sound()}")
print(f"Dog sleeping: {dog.sleep()}")  # Inherited method
print(f"Dog species: {dog.species}")   # Inherited attribute
```

1.2 Inheritance Hierarchies and Relationships

Real-world entities naturally form hierarchical relationships that inheritance can model effectively. These hierarchies represent increasingly specific classifications, where each level adds more detailed characteristics while maintaining the properties of higher levels.

University communities provide excellent examples of inheritance hierarchies. Every member of a university community shares certain basic characteristics (name, ID number, contact information), but different roles have specialized attributes and behaviors. Students have enrollment status and GPAs, employees have salary information and departments, while faculty members have additional research areas and tenure status.

Consider a shape hierarchy in computer graphics. All shapes share common properties like position and color, but specific shapes like circles, rectangles, and triangles have unique calculation methods for area and perimeter while maintaining their shared shape characteristics.

```python
# Demonstrating inheritance hierarchy with shapes
class Shape:
    def __init__(self, x, y, color):
        self.x = x              # Common position attribute
        self.y = y              # Common position attribute
        self.color = color      # Common visual attribute

    def move(self, dx, dy):
        self.x += dx            # Common behavior
        self.y += dy            # All shapes can move

    def area(self):
        raise NotImplementedError("Subclasses must implement area()")

class Circle(Shape):
    def __init__(self, x, y, color, radius):
        super().__init__(x, y, color)  # Initialize parent attributes
        self.radius = radius           # Circle-specific attribute

    def area(self):
        return 3.14159 * self.radius ** 2  # Circle-specific calculation

class Rectangle(Shape):
    def __init__(self, x, y, color, width, height):
        super().__init__(x, y, color)  # Initialize parent attributes
        self.width = width             # Rectangle-specific attributes
        self.height = height

    def area(self):
        return self.width * self.height  # Rectangle-specific calculation

```

```python
# Demonstrating the hierarchy in action
circle = Circle(0, 0, "red", 5)
rectangle = Rectangle(10, 10, "blue", 4, 6)

print(f"Circle area: {circle.area():.2f}")
print(f"Rectangle area: {rectangle.area()}")

# Both shapes can use inherited methods
circle.move(5, 5)
rectangle.move(-2, 3)
print(f"Circle position: ({circle.x}, {circle.y})")
print(f"Rectangle position: ({rectangle.x}, {rectangle.y})")
```

1.3 Method Overriding and the super() Function

Method overriding allows derived classes to provide specialized implementations of base class methods while the super() function enables calling the parent implementation. This mechanism provides both specialization and code reuse, allowing derived classes to extend or modify parent behavior without completely reimplementing functionality.

Think of method overriding like regional variations of a recipe. The basic pizza recipe provides the foundation (dough, sauce, cheese), but New York-style and Chicago-style variations override specific steps while still following the core recipe structure. The super() function is like referring back to the original recipe while adding your regional modifications.

This pattern is extremely powerful because it allows incremental specialization. A base class defines default behavior, and derived classes can choose which methods to override for their specific needs while inheriting everything else unchanged.

```python
# Base Employee class defining common behavior
from decimal import Decimal

class Employee:
    def __init__(self, first_name, last_name, ssn):
        self.first_name = first_name
        self.last_name = last_name
        self.ssn = ssn

    def earnings(self):
        return Decimal('0.00')  # Base implementation

    def __repr__(self):
        return f"Employee: {self.first_name} {self.last_name}\nSSN: {
            self.ssn}"
```

The base Employee class establishes the foundation for all employee types. Every employee has a name and social security number, and all employees can calculate earnings (though the base implementation returns zero). This creates a common interface that derived classes will specialize.

```python
# SalariedEmployee demonstrates basic method overriding
class SalariedEmployee(Employee):
    def __init__(self, first_name, last_name, ssn, weekly_salary):
        super().__init__(first_name, last_name, ssn)  # Call parent
            constructor
        self.weekly_salary = weekly_salary

    def earnings(self):  # Override parent method
        return self.weekly_salary  # Specialized calculation
```

```python

    def __repr__(self):  # Override string representation
        return (super().__repr__() +  # Use parent implementation
                f"\nWeekly salary: ${self.weekly_salary:.2f}")
```

SalariedEmployee extends Employee by adding a weekly salary attribute and overriding the earnings() method to return that salary. Notice how super() is used to call the parent constructor and to extend (rather than replace) the parent's string representation.

```python
# CommissionEmployee shows different specialization approach
class CommissionEmployee(Employee):
    def __init__(self, first_name, last_name, ssn, gross_sales,
        commission_rate):
        super().__init__(first_name, last_name, ssn)  # Call parent
            constructor
        self.gross_sales = gross_sales
        self.commission_rate = commission_rate

    def earnings(self):  # Override with different calculation
        return self.gross_sales * self.commission_rate

    def __repr__(self):  # Override string representation
        return (super().__repr__() +  # Use parent implementation
                f"\nGross sales: ${self.gross_sales:.2f}" +
                f"\nCommission rate: {self.commission_rate:.2f}")
```

CommissionEmployee demonstrates how different derived classes can implement the same interface (earnings()) in completely different ways. While SalariedEmployee returns a fixed amount, CommissionEmployee calculates earnings based on sales performance.

```python
# Demonstrating polymorphic behavior with different employee types
salary_emp = SalariedEmployee("John", "Smith", "111-11-1111", Decimal('
    800.00'))
commission_emp = CommissionEmployee("Sue", "Jones", "222-22-2222",
                                    Decimal('10000.00'), Decimal('0.06'))

print("Salaried Employee:")
print(salary_emp)
print(f"Earnings: ${salary_emp.earnings():.2f}\n")

print("Commission Employee:")
print(commission_emp)
print(f"Earnings: ${commission_emp.earnings():.2f}")
```

2 Part 2: Polymorphism in Action

2.1 Understanding Polymorphic Behavior

Polymorphism allows objects of different types to be treated uniformly through common interfaces, enabling flexible and extensible code design. The word "polymorphism" comes from Greek meaning "many forms," and it represents one of the most powerful features of object-oriented programming.

Consider a universal remote control that can operate different devices (TV, DVD player, stereo system) using the same buttons. You press the "play" button, and each device responds appropriately based on its type. The remote doesn't need to know the specific implementation details of each device; it just needs to know that each device can respond to the "play" command.

In programming, polymorphism works similarly. You can have a collection of different objects and call the same method on each one, with each object responding according to its specific type. This creates code that is both flexible and easy to maintain because you can add new types without modifying existing code.

```python
# Polymorphism demonstration with different employee types
def process_employees(employee_list):
    """Process a list of employees polymorphically"""
    total_payroll = Decimal('0.00')

    print("=== EMPLOYEE PAYROLL PROCESSING ===")
    for employee in employee_list:
        # Same method call works for all employee types
        earnings = employee.earnings()  # Polymorphic method call
        total_payroll += earnings

        print(f"\n{employee}")  # Polymorphic string representation
        print(f"Earnings: ${earnings:.2f}")

    print(f"\nTotal Payroll: ${total_payroll:.2f}")

# Create different types of employees
employees = [
    SalariedEmployee("Alice", "Johnson", "333-33-3333", Decimal('
        1000.00')),
    CommissionEmployee("Bob", "Wilson", "444-44-4444",
                       Decimal('15000.00'), Decimal('0.08')),
    SalariedEmployee("Carol", "Davis", "555-55-5555", Decimal('1200.00'
        ))
]

# Process all employees polymorphically
# The same code works for all employee types
process_employees(employees)
```

2.2 Testing Inheritance Relationships

Python provides built-in functions to test inheritance relationships and object types. The isinstance() function determines whether an object has an "is-a" relationship with a specific type, while issubclass() determines whether one class is derived from another.

These functions are essential for writing robust polymorphic code because they allow you to verify type relationships at runtime. This is particularly useful when you need to handle objects differently based on their specific types while still maintaining polymorphic behavior.

Understanding these relationships helps ensure your code works correctly with inheritance hierarchies and can make intelligent decisions about how to process different object types.

```python
# Testing inheritance relationships and type checking
def analyze_employee_relationships():
    """Demonstrate isinstance() and issubclass() functions"""

    # Create sample objects
    base_emp = Employee("Test", "Employee", "000-00-0000")
    sal_emp = SalariedEmployee("Salary", "Worker", "111-11-1111",
        Decimal('800.00'))

    print("=== INHERITANCE RELATIONSHIP TESTING ===")

```

```python
    # Test isinstance() - object to type relationships
    print(f"sal_emp is Employee: {isinstance(sal_emp, Employee)}")
    print(f"sal_emp is SalariedEmployee: {isinstance(sal_emp, SalariedEmployee)}")
    print(f"base_emp is SalariedEmployee: {isinstance(base_emp, SalariedEmployee)}")

    # Test issubclass() - class to class relationships
    print(f"SalariedEmployee is subclass of Employee: {issubclass(SalariedEmployee, Employee)}")
    print(f"Employee is subclass of SalariedEmployee: {issubclass(Employee, SalariedEmployee)}")

    # Demonstrate polymorphic type checking
    def process_if_employee(obj):
        if isinstance(obj, Employee):
            print(f"Processing employee: {obj.first_name} {obj.last_name}")
            print(f"Earnings: ${obj.earnings():.2f}")
        else:
            print("Object is not an employee")

    print("\n=== POLYMORPHIC TYPE CHECKING ===")
    process_if_employee(sal_emp)
    process_if_employee("Not an employee")

# Run the relationship analysis
analyze_employee_relationships()
```

3 Part 3: Duck Typing and Dynamic Polymorphism

3.1 Understanding Duck Typing Philosophy

Duck typing represents Python's dynamic approach to polymorphism where an object's suitability is determined by its methods and properties rather than its type. The name comes from the saying "If it looks like a duck and quacks like a duck, it must be a duck." In programming terms, if an object has the required methods, it can be used regardless of its actual class hierarchy.

This concept is particularly powerful because it allows objects from completely unrelated class hierarchies to work together as long as they provide the expected interface. It's like being able to use any device that has a USB port with your computer, regardless of whether it's a keyboard, mouse, or storage device—what matters is the interface, not the specific device type.

Duck typing enables more flexible code design because you don't need to plan inheritance hierarchies in advance. Any object that provides the required methods can participate in your system, making your code more adaptable and extensible.

```python
# Duck typing demonstration with file-like objects
class StringLogger:
    """A simple string-based logger that behaves like a file"""
    def __init__(self):
        self._content = []

    def write(self, text):
        """Duck typing method - acts like file.write()"""
        self._content.append(text)

```

```python
    def read(self):
        """Duck typing method - acts like file.read()"""
        return ''.join(self._content)

    def close(self):
        """Duck typing method - acts like file.close()"""
        print("Logger closed")

class NetworkLogger:
    """A network-based logger that also behaves like a file"""
    def __init__(self, server):
        self.server = server
        self._buffer = []

    def write(self, text):
        """Duck typing method - same interface as file"""
        self._buffer.append(f"[{self.server}] {text}")

    def read(self):
        """Duck typing method - same interface as file"""
        return '\n'.join(self._buffer)

    def close(self):
        """Duck typing method - same interface as file"""
        print(f"Network connection to {self.server} closed")

def log_messages(logger):
    """Function that works with any file-like object"""
    # This function doesn't care about the specific type
    # It only cares that the object has write() and close() methods
    logger.write("System startup")
    logger.write("User login: john_doe")
    logger.write("Processing complete")
    logger.close()

# Duck typing in action - same function works with different types
print("=== DUCK TYPING DEMONSTRATION ===")

string_logger = StringLogger()
network_logger = NetworkLogger("logging.server.com")

print("Using StringLogger:")
log_messages(string_logger)
print(f"Content: {string_logger.read()}")

print("\nUsing NetworkLogger:")
log_messages(network_logger)
print(f"Content: {network_logger.read()}")
```

3.2 Duck Typing with Employee Systems

Duck typing becomes particularly powerful when you want to extend existing systems with new functionality. You can create objects that work with existing polymorphic code without inheriting from specific base classes, as long as they provide the expected interface.

This approach is especially useful when integrating third-party components or creating mock objects for testing. The flexibility of duck typing allows these objects to participate in existing workflows without requiring changes to the underlying system architecture.

```python
# Duck typing with employee-like objects
class Contractor:
    """A contractor that works like an employee but isn't one"""
    def __init__(self, name, hourly_rate, hours_worked):
        self.name = name
        self.hourly_rate = hourly_rate
        self.hours_worked = hours_worked

    def earnings(self):
        """Duck typing - same interface as Employee.earnings()"""
        return self.hourly_rate * self.hours_worked

    def __repr__(self):
        """Duck typing - same interface as Employee.__repr__()"""
        return f"Contractor: {self.name}\nHourly rate: ${self.hourly_rate:.2f}\nHours: {self.hours_worked}"

class Volunteer:
    """A volunteer that works like an employee for processing"""
    def __init__(self, name, cause):
        self.name = name
        self.cause = cause

    def earnings(self):
        """Duck typing - volunteers earn $0 but have same interface"""
        return Decimal('0.00')

    def __repr__(self):
        """Duck typing - same interface as Employee.__repr__()"""
        return f"Volunteer: {self.name}\nCause: {self.cause}"

# Function that works with any object that has earnings() method
def calculate_total_compensation(workers):
    """Calculate total compensation for any workers with earnings() method"""
    total = Decimal('0.00')

    print("=== WORKER COMPENSATION ANALYSIS ===")
    for worker in workers:
        # Duck typing - we don't care about the specific type
        # We only care that the object has earnings() and __repr__() methods
        compensation = worker.earnings()
        total += compensation

        print(f"\n{worker}")
        print(f"Compensation: ${compensation:.2f}")

    print(f"\nTotal Compensation: ${total:.2f}")

# Mix different types of workers - duck typing makes this work
all_workers = [
    SalariedEmployee("Alice", "Johnson", "111-11-1111", Decimal('1000.00')),
    Contractor("Bob", "Freelancer", Decimal('50.00'), 40),
    Volunteer("Carol", "Community Helper", "Local Food Bank"),
    CommissionEmployee("Dave", "Sales", "222-22-2222",
```

```
                        Decimal('8000.00'), Decimal('0.10'))
]

# Same function works with all types due to duck typing
calculate_total_compensation(all_workers)
```

4 Part 4: Operator Overloading

4.1 Understanding Special Methods

Operator overloading allows custom classes to work with Python operators by implementing special methods that define operator behavior. These special methods, identifiable by their double underscore names (__add__, __eq__, __lt__), enable objects to use natural syntax like mathematical operations and comparisons.

Think of operator overloading like teaching your custom objects to speak Python's built-in language. Just as you can add numbers with the + operator or compare them with ==, your custom objects can participate in these same operations by defining what these operators mean for your specific data type.

This capability makes custom classes feel like natural extensions of Python's built-in types. Mathematical vectors can use + for vector addition, custom point objects can use ¡ for distance-based comparisons, and complex numbers can use all mathematical operators just like integers and floats.

```
# Vector class demonstrating operator overloading
import math

class Vector2D:
    """A 2D vector class with operator overloading"""

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        """Overload + operator for vector addition"""
        if isinstance(other, Vector2D):
            return Vector2D(self.x + other.x, self.y + other.y)
        return NotImplemented

    def __sub__(self, other):
        """Overload - operator for vector subtraction"""
        if isinstance(other, Vector2D):
            return Vector2D(self.x - other.x, self.y - other.y)
        return NotImplemented

    def __mul__(self, scalar):
        """Overload * operator for scalar multiplication"""
        if isinstance(scalar, (int, float)):
            return Vector2D(self.x * scalar, self.y * scalar)
        return NotImplemented

    def __eq__(self, other):
        """Overload == operator for vector equality"""
        if isinstance(other, Vector2D):
            return self.x == other.x and self.y == other.y
        return False
```

```python

    def __lt__(self, other):
        """Overload < operator for magnitude comparison"""
        if isinstance(other, Vector2D):
            return self.magnitude() < other.magnitude()
        return NotImplemented

    def magnitude(self):
        """Calculate vector magnitude (length)"""
        return math.sqrt(self.x**2 + self.y**2)

    def __repr__(self):
        """String representation of vector"""
        return f"Vector2D({self.x}, {self.y})"

# Demonstrating operator overloading in action
print("=== VECTOR OPERATOR OVERLOADING ===")

v1 = Vector2D(3, 4)
v2 = Vector2D(1, 2)
v3 = Vector2D(3, 4)

print(f"v1 = {v1}")
print(f"v2 = {v2}")
print(f"v3 = {v3}")

# Vector addition using overloaded +
result_add = v1 + v2
print(f"\nv1 + v2 = {result_add}")

# Vector subtraction using overloaded -
result_sub = v1 - v2
print(f"v1 - v2 = {result_sub}")

# Scalar multiplication using overloaded *
result_mul = v1 * 2
print(f"v1 * 2 = {result_mul}")

# Vector comparison using overloaded ==
print(f"\nv1 == v2: {v1 == v2}")
print(f"v1 == v3: {v1 == v3}")

# Magnitude comparison using overloaded <
print(f"v2 < v1: {v2 < v1}")
print(f"Magnitude v1: {v1.magnitude():.2f}")
print(f"Magnitude v2: {v2.magnitude():.2f}")
```

4.2 Complex Number Implementation

Complex numbers provide an excellent example of comprehensive operator overloading because they require mathematical operations that mirror real number arithmetic while handling both real and imaginary components. This demonstrates how operator overloading can make custom types behave like built-in Python types.

Mathematical operations on complex numbers follow specific rules: addition combines real parts and imaginary parts separately, multiplication requires the distributive property with special handling of i^2, and equality requires both components to match. Implementing these operations shows how operator overloading can capture domain-specific mathematical behavior.

```
# Complex number class with comprehensive operator overloading
class Complex:
    """Complex number class with mathematical operations"""

    def __init__(self, real, imaginary):
        self.real = real
        self.imaginary = imaginary

    def __add__(self, other):
        """Complex addition: (a+bi) + (c+di) = (a+c) + (b+d)i"""
        if isinstance(other, Complex):
            return Complex(self.real + other.real,
                           self.imaginary + other.imaginary)
        elif isinstance(other, (int, float)):
            return Complex(self.real + other, self.imaginary)
        return NotImplemented

    def __sub__(self, other):
        """Complex subtraction: (a+bi) - (c+di) = (a-c) + (b-d)i"""
        if isinstance(other, Complex):
            return Complex(self.real - other.real,
                           self.imaginary - other.imaginary)
        elif isinstance(other, (int, float)):
            return Complex(self.real - other, self.imaginary)
        return NotImplemented

    def __mul__(self, other):
        """Complex multiplication: (a+bi)(c+di) = (ac-bd) + (ad+bc)i"""
        if isinstance(other, Complex):
            real_part = self.real * other.real - self.imaginary * other.imaginary
            imag_part = self.real * other.imaginary + self.imaginary * other.real
            return Complex(real_part, imag_part)
        elif isinstance(other, (int, float)):
            return Complex(self.real * other, self.imaginary * other)
        return NotImplemented

    def __eq__(self, other):
        """Complex equality: both real and imaginary parts must match"""
        if isinstance(other, Complex):
            return (self.real == other.real and
                    self.imaginary == other.imaginary)
        elif isinstance(other, (int, float)):
            return self.real == other and self.imaginary == 0
        return False

    def __abs__(self):
        """Complex magnitude: |a+bi| = sqrt(a^2 + b^2)"""
        return math.sqrt(self.real**2 + self.imaginary**2)

    def __repr__(self):
        """String representation of complex number"""
        if self.imaginary >= 0:
            return f"({self.real} + {self.imaginary}i)"
        else:
```

```
            return f"({self.real} - {abs(self.imaginary)}i)"

# Demonstrating complex number operations
print("=== COMPLEX NUMBER OPERATIONS ===")

z1 = Complex(3, 4)
z2 = Complex(2, -1)
z3 = Complex(3, 4)

print(f"z1 = {z1}")
print(f"z2 = {z2}")
print(f"z3 = {z3}")

# Complex arithmetic operations
print(f"\nz1 + z2 = {z1 + z2}")
print(f"z1 - z2 = {z1 - z2}")
print(f"z1 * z2 = {z1 * z2}")

# Operations with real numbers
print(f"\nz1 + 5 = {z1 + 5}")
print(f"z1 * 3 = {z1 * 3}")

# Comparison and magnitude
print(f"\nz1 == z2: {z1 == z2}")
print(f"z1 == z3: {z1 == z3}")
print(f"|z1| = {abs(z1):.2f}")
print(f"|z2| = {abs(z2):.2f}")
```

5 Part 5: String Fundamentals

5.1 String Creation and Initialization

Strings are one of the most fundamental data types in Python, representing text data that your programs can manipulate and process. Just like numbers store mathematical values, strings store textual information - from single characters to entire documents. Think of a string as a necklace where each bead is a character, and Python gives you powerful tools to work with this necklace in various ways.

In Python, you can create strings using single quotes, double quotes, or triple quotes. Each method has its purpose, and understanding when to use each one will make your code more readable and easier to write. The flexibility in string creation is one of Python's strengths, allowing you to handle text data in whatever form it appears.

```
# Different ways to create strings
name = 'Alice'  # Single quotes for simple strings
message = "It's a beautiful day!"  # Double quotes when string contains apostrophes
paragraph = """This is a multi-line string.
It can span multiple lines.
Perfect for long text or documentation."""  # Triple quotes for multi-line

# Special characters in strings using backslash
file_path = "C:\\Users\\Alice\\Documents"  # Backslash for special characters
quote = "She said, \"Hello, World!\""  # Escaping quotes inside strings
new_line = "First Line\nSecond Line"  # \n creates a new line
```

```python

# Creating empty strings and string from other types
empty = ""  # Empty string
number_string = str(42)  # Converting number to string: "42"
float_string = str(3.14159)  # Converting float to string: "3.14159"

print(f"Name: {name}")
print(f"Message: {message}")
print(f"Paragraph:\n{paragraph}")
print(f"Quote: {quote}")
print(f"With newline:\n{new_line}")
```

5.2 String Indexing and Slicing

Strings in Python are sequences of characters, and each character has a position called an index. Think of it like a row of houses on a street - each house has an address number. In Python strings, these addresses start at 0 for the first character. You can also use negative numbers to count from the end, where -1 is the last character.

Slicing is a powerful feature that lets you extract portions of a string. It's like taking a photograph of just part of a landscape - you specify where to start and where to end, and Python gives you just that section. The syntax uses square brackets with colons to separate the start, stop, and step values.

```python
# String indexing - accessing individual characters
word = "Python"
first_char = word[0]  # 'P' - first character
last_char = word[-1]  # 'n' - last character
middle_char = word[2]  # 't' - third character (index 2)

print(f"Word: {word}")
print(f"First character: {first_char}")
print(f"Last character: {last_char}")
print(f"Character at index 2: {middle_char}")

# String slicing - extracting substrings
phrase = "Hello, Python World!"

# Basic slicing [start:end] - end is exclusive
greeting = phrase[0:5]  # "Hello"
language = phrase[7:13]  # "Python"

# Omitting indices
start_to_comma = phrase[:5]  # "Hello" - from beginning
from_comma = phrase[7:]  # "Python World!" - to end
complete_copy = phrase[:]  # Full copy of string

# Using step value [start:end:step]
every_second = phrase[::2]  # "Hlo yhnWrd" - every 2nd character
reversed_string = phrase[::-1]  # "!dlroW nohtyP ,olleH" - reverse string

print(f"\nOriginal: '{phrase}'")
print(f"phrase[0:5]: '{greeting}'")
print(f"phrase[7:13]: '{language}'")
print(f"phrase[:5]: '{start_to_comma}'")
print(f"phrase[7:]: '{from_comma}'")
print(f"Every 2nd char: '{every_second}'")
```

```python
print(f"Reversed: '{reversed_string}'")
```

5.3 Essential String Methods

Python provides numerous built-in methods to work with strings efficiently. These methods are like specialized tools in a toolkit - each designed for a specific text manipulation task. String methods don't modify the original string (strings are immutable); instead, they return new strings with the desired changes.

Understanding these methods is crucial for text processing, data cleaning, and user input handling. They save you from writing complex code for common operations and make your programs more readable and maintainable.

```python
# Case conversion methods
text = "Hello Python World"
upper_text = text.upper()  # "HELLO PYTHON WORLD"
lower_text = text.lower()  # "hello python world"
title_text = text.title()  # "Hello Python World"
swap_text = text.swapcase()  # "hELLO pYTHON wORLD"

print("Case conversions:")
print(f"Original: {text}")
print(f"Upper: {upper_text}")
print(f"Lower: {lower_text}")
print(f"Title: {title_text}")
print(f"Swapcase: {swap_text}")

# Searching and checking methods
sentence = "Python is amazing. Python is powerful."

# Finding substrings
first_python = sentence.find("Python")  # 0 - index of first occurrence
last_python = sentence.rfind("Python")  # 19 - index of last occurrence
count_python = sentence.count("Python")  # 2 - number of occurrences

# Checking string properties
starts_with_python = sentence.startswith("Python")  # True
ends_with_period = sentence.endswith(".")  # True
is_alphanumeric = "Python3".isalnum()  # True
is_alphabetic = "Python".isalpha()  # True
is_numeric = "12345".isdigit()  # True

print(f"\nSearching in: '{sentence}'")
print(f"First 'Python' at index: {first_python}")
print(f"Last 'Python' at index: {last_python}")
print(f"'Python' appears {count_python} times")
print(f"Starts with 'Python': {starts_with_python}")
print(f"Ends with '.': {ends_with_period}")

# String modification methods
messy_input = "   Hello,   World!   "
clean_input = messy_input.strip()  # Remove leading/trailing spaces
no_hello = clean_input.replace("Hello", "Hi")  # Replace text

# Splitting and joining
data = "apple,banana,orange,grape"
fruits = data.split(",")  # ['apple', 'banana', 'orange', 'grape']
rejoined = " | ".join(fruits)  # "apple | banana | orange | grape"
```

```python

print(f"\nOriginal messy input: '{messy_input}'")
print(f"After strip(): '{clean_input}'")
print(f"After replace(): '{no_hello}'")
print(f"Split result: {fruits}")
print(f"Joined with ' | ': '{rejoined}'")
```

5.4 String Formatting Techniques

String formatting is essential for creating dynamic text output that combines static text with variable data. Python offers multiple formatting approaches, each with its own advantages. Modern Python code typically uses f-strings (formatted string literals) for their readability and performance, but understanding all methods helps you work with existing code and choose the best approach for your specific needs.

Think of string formatting like filling in a form template - you have the structure of the text with blank spaces where variable information gets inserted. This is incredibly useful for creating user messages, generating reports, or building any text that combines fixed and variable content.

```python
# Method 1: f-strings (Python 3.6+) - Recommended
name = "Alice"
age = 25
height = 5.6

# Basic f-string formatting
basic = f"My name is {name} and I am {age} years old."
print(f"Basic f-string: {basic}")

# F-strings with expressions and formatting
calc = f"In 5 years, {name} will be {age + 5} years old."
formatted_height = f"{name} is {height:.1f} feet tall."  # One decimal place
percentage = 0.875
formatted_percent = f"Test score: {percentage:.1%}"  # As percentage

print(f"With calculation: {calc}")
print(f"Formatted number: {formatted_height}")
print(f"As percentage: {formatted_percent}")

# Method 2: .format() method - Still widely used
template = "Hello, {}! You have {} new messages."
filled = template.format("Bob", 3)

# Named placeholders
named_template = "Dear {customer}, your order #{order_id} is ready."
filled_named = named_template.format(customer="Carol", order_id=12345)

print(f"\nUsing .format(): {filled}")
print(f"Named placeholders: {filled_named}")

# Method 3: % formatting (older style)
old_style = "Hello, %s! You scored %d points." % ("David", 95)
float_format = "Pi is approximately %.2f" % 3.14159  # Two decimal places

print(f"\nOld style %: {old_style}")
print(f"Float formatting: {float_format}")
```

```python

# Alignment and padding in f-strings
items = [("Apple", 1.50), ("Banana", 0.75), ("Orange", 2.00)]
print("\nFormatted receipt:")
print(f"{'Item':<15} {'Price':>7}")  # Left and right alignment
print("-" * 23)
for item, price in items:
    print(f"{item:<15} ${price:>6.2f}")
```

5.5 String Immutability and Operations

One crucial concept to understand about Python strings is that they are immutable - once created, they cannot be changed. This might seem limiting at first, but it's actually a powerful feature that makes strings safer to use and more efficient in many situations. When you perform operations that seem to modify a string, Python actually creates a new string with the changes.

Think of string immutability like a printed book - you can't change the words on a printed page, but you can create a new edition with different content. This concept affects how you work with strings and is important for writing efficient code.

```python
# Demonstrating string immutability
original = "Hello"
# original[0] = "h"  # This would cause an error!

# Instead, create new strings
lowercase = original.lower()  # Creates new string "hello"
print(f"Original: {original}")  # Still "Hello"
print(f"Lowercase: {lowercase}")  # New string "hello"

# String concatenation creates new strings
first = "Hello"
second = " World"
combined = first + second  # New string "Hello World"
print(f"Combined: {combined}")

# Efficient string building for multiple concatenations
# Inefficient way (creates many intermediate strings)
result = ""
for i in range(5):
    result += str(i) + " "  # Creates new string each time

# Efficient way using join
numbers = [str(i) for i in range(5)]
efficient_result = " ".join(numbers)

print(f"Loop result: '{result.strip()}'")
print(f"Join result: '{efficient_result}'")

# String multiplication and membership testing
separator = "-" * 20  # Creates "--------------------"
contains_hello = "Hello" in "Hello, World!"  # True
contains_bye = "Bye" in "Hello, World!"  # False

print(f"\nSeparator: {separator}")
print(f"'Hello' in 'Hello, World!': {contains_hello}")
print(f"'Bye' in 'Hello, World!': {contains_bye}")

# Comparing strings
```

```python
str1 = "apple"
str2 = "Apple"
str3 = "apple"

# Case-sensitive comparison
equal_exact = str1 == str2  # False (different case)
equal_same = str1 == str3  # True

# Case-insensitive comparison
equal_ignore_case = str1.lower() == str2.lower()  # True

print(f"\n'{str1}' == '{str2}': {equal_exact}")
print(f"'{str1}' == '{str3}': {equal_same}")
print(f"Case-insensitive comparison: {equal_ignore_case}")
```

6 Part 6: Advanced String Operations

6.1 Escape Sequences and Special Characters

When working with strings, you'll often need to include special characters that can't be typed directly or have special meaning in Python. Escape sequences allow you to represent these characters using a backslash followed by a character code. Understanding escape sequences is crucial for handling file paths, formatting output, and working with text data from various sources.

Think of escape sequences as secret codes that tell Python to interpret characters in a special way. Just like in writing where we use punctuation marks to indicate pauses or emphasis, escape sequences let us include formatting instructions within our strings. The backslash acts as an escape character, signaling that the next character should be treated specially.

```python
# Common escape sequences
print("=== ESCAPE SEQUENCES ===")

# Newline and tab
text = "First Line\nSecond Line\tWith a tab"
print(text)

# Quotes within strings
single = 'It\'s a beautiful day!'
double = "She said, \"Hello, World!\""
print(single)
print(double)

# Backslash itself needs escaping
file_path = "C:\\Users\\Alice\\Documents\\file.txt"
print(f"Windows path: {file_path}")

# Unicode characters using \u
hearts = "I \u2665 Python!"  # Heart symbol
smiley = "Happy coding! \u263A"  # Smiley face
print(hearts)
print(smiley)

# Common escape sequences reference
escape_demo = """
Common Escape Sequences:
\\n - Newline
```

```
\\t - Tab
\\r - Carriage return
\\\\ - Backslash
\\' - Single quote
\\" - Double quote
\\b - Backspace
\\f - Form feed
\\v - Vertical tab
\\0 - Null character
"""
print(escape_demo)

# Demonstrating carriage return
import time
print("Progress: ", end="")
for i in range(101):
    print(f"\rProgress: {i}%", end="", flush=True)
    # Simulating work (comment out time.sleep in notebook)
    # time.sleep(0.01)
print("\nComplete!")
```

6.2 Raw Strings and String Literals

Raw strings, denoted by an 'r' prefix, treat backslashes as literal characters rather than escape characters. This is particularly useful when working with regular expressions, file paths, or any text where backslashes are common. Raw strings save you from the "backslash plague" where you'd otherwise need to double every backslash.

The 'r' prefix tells Python to read the string exactly as written, without processing escape sequences. It's like telling Python to put on reading glasses and see every character exactly as it appears, without trying to interpret special meanings. This makes raw strings invaluable for certain applications.

```
# Raw strings demonstration
print("=== RAW STRINGS ===")

# Compare normal vs raw strings
normal_path = "C:\\Users\\Alice\\Documents\\newfile.txt"
raw_path = r"C:\Users\Alice\Documents\newfile.txt"

print(f"Normal string: {normal_path}")
print(f"Raw string: {raw_path}")
print(f"Are they equal? {normal_path == raw_path}")

# Raw strings with quotes
regex_pattern = r"^\d{3}-\d{3}-\d{4}$"  # Phone number pattern
print(f"Regex pattern: {regex_pattern}")

# Where raw strings don't work - can't end with backslash
# invalid = r"This ends with \"  # SyntaxError!
# Use normal string or concatenation for this case
valid = r"This ends with" + "\\"
print(valid)

# Multi-line raw strings
sql_query = r"""
SELECT *
FROM users
```

```
WHERE email LIKE '%@example.com'
AND created_date >= '2024-01-01'
"""
print(f"SQL Query:{sql_query}")

# Combining string prefixes
# f-strings can't be combined with r-strings in Python < 3.12
name = "Alice"
# Can't do: fr"C:\Users\{name}\Documents"
# Instead:
path = rf"C:\Users\{name}\Documents"  # Works in Python 3.12+
# Or for older versions:
path = r"C:\Users" + f"\\{name}" + r"\Documents"
print(f"User path: {path}")
```

6.3 String Comparison and Sorting

String comparison in Python follows lexicographical order, similar to how words are arranged in a dictionary. Understanding how Python compares strings is essential for sorting data, validating input, and implementing search algorithms. Python compares strings character by character using their Unicode values.

When comparing strings, Python acts like a very precise librarian organizing books. It looks at each character position from left to right, comparing the Unicode values. This means 'Z' comes before 'a' because uppercase letters have lower Unicode values than lowercase letters. This behavior impacts how you sort and compare text data.

```
# String comparison basics
print("=== STRING COMPARISON ===")

# Basic comparisons
print(f"'apple' < 'banana': {'apple' < 'banana'}")
print(f"'Apple' < 'apple': {'Apple' < 'apple'}")  # Uppercase first!
print(f"'app' < 'apple': {'app' < 'apple'}")  # Shorter string first

# Demonstrating character-by-character comparison
str1 = "abc"
str2 = "abd"
print(f"\n'{str1}' vs '{str2}':")
for i, (c1, c2) in enumerate(zip(str1, str2)):
    print(f"Position {i}: '{c1}' (ord={ord(c1)}) vs '{c2}' (ord={ord(c2)})")

# Sorting strings
fruits = ["banana", "Apple", "cherry", "apricot"]
print(f"\nOriginal list: {fruits}")

# Default sort (case-sensitive)
sorted_default = sorted(fruits)
print(f"Default sort: {sorted_default}")

# Case-insensitive sort
sorted_case_insensitive = sorted(fruits, key=str.lower)
print(f"Case-insensitive: {sorted_case_insensitive}")

# Custom sorting
names = ["Alice Smith", "Bob Jones", "Charlie Brown", "Alice Johnson"]

```

```python
# Sort by last name
sorted_by_last = sorted(names, key=lambda x: x.split()[-1])
print(f"\nSorted by last name: {sorted_by_last}")

# Sort by length, then alphabetically
words = ["pie", "apple", "ai", "application", "app", "a"]
sorted_complex = sorted(words, key=lambda x: (len(x), x))
print(f"By length, then alpha: {sorted_complex}")

# Finding min and max strings
print(f"\nMin fruit: {min(fruits)}")  # Capital letters come first
print(f"Max fruit: {max(fruits)}")
print(f"Min (case-insensitive): {min(fruits, key=str.lower)}")
```

6.4 String Encoding and Unicode

In our globalized world, handling text in multiple languages is essential. Python 3 uses Unicode by default, allowing you to work with text from any language. Understanding encoding and decoding helps you handle international text, read files from different sources, and communicate with systems using various character encodings.

Character encoding is like translating between different writing systems. Just as Morse code represents letters as dots and dashes, character encodings represent text characters as numbers that computers can store. Unicode is the universal translator that assigns a unique number to every character in every language, while encodings like UTF-8 determine how these numbers are stored as bytes.

```python
# Unicode and encoding demonstration
print("=== UNICODE AND ENCODING ===")

# Unicode strings with various languages
greetings = {
    "English": "Hello",
    "Spanish": "Hola",
    "French": "Bonjour",
    "German": "Guten Tag",
    "Japanese": "konnichiwa",
    "Arabic": "marhaban",
    "Russian": "privet",
    "Emoji": "wave + globe"
}

for lang, greeting in greetings.items():
    print(f"{lang}: {greeting}")

# Encoding strings to bytes
text = "Hello, World!"  # Mixed English and other languages
print(f"\nOriginal text: {text}")

# Different encodings
utf8_bytes = text.encode('utf-8')
print(f"UTF-8 bytes: {utf8_bytes}")
print(f"UTF-8 length: {len(utf8_bytes)} bytes")

# Some characters can't be encoded in ASCII
try:
    ascii_bytes = text.encode('ascii')
except UnicodeEncodeError as e:
```

```python
    print(f"ASCII encoding failed: {e}")

# Using errors parameter
ascii_ignore = text.encode('ascii', errors='ignore')
ascii_replace = text.encode('ascii', errors='replace')
print(f"ASCII (ignore): {ascii_ignore}")
print(f"ASCII (replace): {ascii_replace}")

# Decoding bytes back to strings
utf8_decoded = utf8_bytes.decode('utf-8')
print(f"\nDecoded from UTF-8: {utf8_decoded}")

# Unicode code points
for char in "ABC":
    print(f"'{char}': U+{ord(char):04X} (decimal: {ord(char)})")

# Creating characters from code points
heart = chr(0x2764)  # Heavy black heart
star = chr(0x2B50)   # Star
print(f"\nHeart symbol (U+2764), Star symbol (U+2B50)")
```

7 Part 7: String Algorithms and Applications

7.1 Common String Algorithms

String algorithms form the foundation of text processing and are used in countless applications from search engines to DNA analysis. Understanding these basic algorithms helps you solve common programming challenges and provides insights into how text processing works. These algorithms demonstrate important programming concepts like iteration, comparison, and pattern matching.

Learning string algorithms is like learning cooking techniques - once you master the basics like chopping and sautéing, you can combine them to create complex dishes. Similarly, basic string algorithms like checking palindromes or finding patterns can be combined to solve sophisticated text processing problems.

```python
# Common string algorithms
print("=== STRING ALGORITHMS ===")

# Palindrome checker
def is_palindrome(s):
    """Check if string is palindrome (reads same forwards and backwards
       )"""
    # Clean string: remove spaces and convert to lowercase
    clean = ''.join(s.lower().split())
    return clean == clean[::-1]

# Test palindromes
test_palindromes = [
    "racecar",
    "A man a plan a canal Panama",
    "race a car",
    "hello",
    "Madam"
]

print("Palindrome Tests:")
```

```python
for text in test_palindromes:
    result = is_palindrome(text)
    print(f"'{text}': {result}")

# Anagram checker
def are_anagrams(s1, s2):
    """Check if two strings are anagrams (same letters, different order)"""
    # Remove spaces and convert to lowercase
    clean1 = ''.join(s1.lower().split())
    clean2 = ''.join(s2.lower().split())
    # Sort characters and compare
    return sorted(clean1) == sorted(clean2)

# Test anagrams
anagram_pairs = [
    ("listen", "silent"),
    ("evil", "vile"),
    ("hello", "world"),
    ("The Eyes", "They See")
]

print("\nAnagram Tests:")
for word1, word2 in anagram_pairs:
    result = are_anagrams(word1, word2)
    print(f"'{word1}' & '{word2}': {result}")

# Word frequency counter
def count_words(text):
    """Count frequency of each word in text"""
    # Convert to lowercase and split into words
    words = text.lower().split()
    # Count frequencies
    frequency = {}
    for word in words:
        # Remove punctuation from word edges
        cleaned = word.strip('.,!?;:"')
        if cleaned:
            frequency[cleaned] = frequency.get(cleaned, 0) + 1
    return frequency

# Test word frequency
sample_text = """
Python is great. Python is powerful.
I love Python programming!
"""
word_freq = count_words(sample_text)
print("\nWord Frequency:")
for word, count in sorted(word_freq.items(), key=lambda x: x[1], reverse=True):
    print(f"{word}: {count}")

# Find all occurrences of a pattern
def find_all_occurrences(text, pattern):
    """Find all starting positions of pattern in text"""
    positions = []
    start = 0
    while True:
```

```python
        pos = text.find(pattern, start)
        if pos == -1:
            break
        positions.append(pos)
        start = pos + 1
    return positions

text = "The cat in the hat sat on the mat"
pattern = "at"
positions = find_all_occurrences(text, pattern)
print(f"\nPattern '{pattern}' found at positions: {positions}")
```

7.2 Text Processing Patterns

Real-world text processing often involves cleaning, validating, and transforming user input. These patterns are essential for building robust applications that handle text data correctly. Whether you're processing form inputs, parsing configuration files, or analyzing text documents, these patterns provide reliable solutions to common challenges.

Text processing is like being a digital janitor and architect combined - you clean up messy input and restructure it into something useful. These patterns help you handle the imperfect, inconsistent text data that real users provide, transforming it into clean, validated, and properly formatted information your programs can use.

```python
# Text processing patterns
print("=== TEXT PROCESSING PATTERNS ===")

# Input validation and cleaning
def clean_email(email):
    """Clean and validate email address"""
    # Remove whitespace and convert to lowercase
    cleaned = email.strip().lower()
    # Basic validation
    if '@' not in cleaned or '.' not in cleaned.split('@')[1]:
        return None
    return cleaned

# Test email cleaning
test_emails = [
    "  John.Doe@Example.COM  ",
    "invalid.email",
    "user@domain",
    "valid@email.com"
]

print("Email Validation:")
for email in test_emails:
    clean = clean_email(email)
    print(f"'{email}' -> {clean}")

# Phone number formatting
def format_phone(phone):
    """Format phone number to standard format"""
    # Remove all non-digits
    digits = ''.join(c for c in phone if c.isdigit())

    if len(digits) == 10:
        return f"({digits[:3]}) {digits[3:6]}-{digits[6:]}"
```

```python
    elif len(digits) == 11 and digits[0] == '1':
        return f"+1 ({digits[1:4]}) {digits[4:7]}-{digits[7:]}"
    else:
        return "Invalid phone number"

# Test phone formatting
test_phones = [
    "1234567890",
    "123-456-7890",
    "(123) 456-7890",
    "1-123-456-7890",
    "invalid"
]

print("\nPhone Number Formatting:")
for phone in test_phones:
    formatted = format_phone(phone)
    print(f"'{phone}' -> {formatted}")

# CSV parsing (simple version)
def parse_csv_line(line):
    """Parse a simple CSV line (no quotes or escaping)"""
    return [field.strip() for field in line.split(',')]

# Parse simple CSV data
csv_data = """Name, Age, City
Alice, 25, New York
Bob, 30, San Francisco
Charlie, 35, Chicago"""

print("\nCSV Parsing:")
lines = csv_data.strip().split('\n')
headers = parse_csv_line(lines[0])
print(f"Headers: {headers}")

for line in lines[1:]:
    values = parse_csv_line(line)
    record = dict(zip(headers, values))
    print(f"Record: {record}")

# Text wrapping
def wrap_text(text, width=50):
    """Wrap text to specified width"""
    words = text.split()
    lines = []
    current_line = []
    current_length = 0

    for word in words:
        if current_length + len(word) + len(current_line) > width:
            lines.append(' '.join(current_line))
            current_line = [word]
            current_length = len(word)
        else:
            current_line.append(word)
            current_length += len(word)

    if current_line:
```

```python
        lines.append(' '.join(current_line))

    return '\n'.join(lines)

long_text = "Python string processing is powerful and flexible. You can
    use it to clean data, validate input, parse files, and much more.
   The key is understanding the available methods and combining them
   effectively."
print("\nText Wrapping:")
print(wrap_text(long_text, 40))
```

7.3 Real-World String Applications

String processing skills are essential in many real-world applications. From building search features to analyzing log files, from creating user-friendly interfaces to processing data files, strings are everywhere in programming. These examples demonstrate how the string concepts we've learned apply to practical programming tasks you'll encounter regularly.

These applications show how string manipulation is like having a Swiss Army knife for text - each tool serves a specific purpose, but together they can handle almost any text processing challenge. Whether you're building a web scraper, analyzing social media posts, or creating a command-line tool, these patterns will serve you well.

```python
# Real-world string applications
print("=== REAL-WORLD APPLICATIONS ===")

# Simple search engine
def search_documents(documents, query):
    """Simple keyword search in documents"""
    query_words = query.lower().split()
    results = []

    for i, doc in enumerate(documents):
        doc_lower = doc.lower()
        # Count matching words
        matches = sum(1 for word in query_words if word in doc_lower)
        if matches > 0:
            results.append((i, matches, doc[:50] + "..."))

    # Sort by relevance (number of matches)
    results.sort(key=lambda x: x[1], reverse=True)
    return results

# Test search
documents = [
    "Python programming is fun and easy to learn",
    "String processing in Python is powerful",
    "Learn Python to build amazing applications",
    "Data science uses Python extensively"
]

query = "Python programming"
results = search_documents(documents, query)
print(f"Search results for '{query}':")
for idx, score, preview in results:
    print(f"  Doc {idx} (score: {score}): {preview}")

# Password strength checker
```

```python
def check_password_strength(password):
    """Check password strength and return score with feedback"""
    score = 0
    feedback = []

    # Length check
    if len(password) >= 8:
        score += 1
    else:
        feedback.append("Use at least 8 characters")

    if len(password) >= 12:
        score += 1

    # Character type checks
    if any(c.isupper() for c in password):
        score += 1
    else:
        feedback.append("Include uppercase letters")

    if any(c.islower() for c in password):
        score += 1
    else:
        feedback.append("Include lowercase letters")

    if any(c.isdigit() for c in password):
        score += 1
    else:
        feedback.append("Include numbers")

    if any(c in "!@#$%^&*()_+-=[]{}|;:,.<>?" for c in password):
        score += 1
    else:
        feedback.append("Include special characters")

    # Determine strength
    if score <= 2:
        strength = "Weak"
    elif score <= 4:
        strength = "Medium"
    else:
        strength = "Strong"

    return strength, score, feedback

# Test passwords
test_passwords = [
    "password",
    "Password1",
    "MyP@ssw0rd!",
    "MyVeryLongP@ssw0rd123!"
]

print("\nPassword Strength Checker:")
for pwd in test_passwords:
    strength, score, feedback = check_password_strength(pwd)
    print(f"\nPassword: {'*' * len(pwd)}")
    print(f"Strength: {strength} (score: {score}/6)")
```

```python
    if feedback:
        print("Suggestions:", ", ".join(feedback))

# URL slug generator (for blog posts, etc.)
def create_url_slug(title):
    """Convert title to URL-friendly slug"""
    # Convert to lowercase
    slug = title.lower()
    # Replace spaces with hyphens
    slug = slug.replace(' ', '-')
    # Remove special characters
    slug = ''.join(c for c in slug if c.isalnum() or c == '-')
    # Remove multiple consecutive hyphens
    while '--' in slug:
        slug = slug.replace('--', '-')
    # Remove leading/trailing hyphens
    slug = slug.strip('-')
    return slug

# Test slug generation
titles = [
    "My First Blog Post!",
    "10 Python String Tips & Tricks",
    "Why I Love Programming???",
    "C++ vs Python: A Comparison"
]

print("\nURL Slug Generator:")
for title in titles:
    slug = create_url_slug(title)
    print(f"'{title}' -> '{slug}'")
```

7.4 Advanced OOP Principles Mastered

Through this lecture, you have mastered the advanced concepts that transform single-class programming into sophisticated multi-class systems. Inheritance allows you to create specialized classes that extend base functionality while maintaining code reuse. Method overriding provides customization points where derived classes can specialize behavior while using super() to access parent implementations.

Polymorphism enables writing flexible code that works with objects of different types through common interfaces. This capability, combined with duck typing, allows Python objects to work together based on their behavior rather than their inheritance relationships, creating extremely adaptable systems.

Operator overloading makes custom classes integrate seamlessly with Python's built-in syntax, allowing mathematical operations, comparisons, and other operators to work naturally with your custom types. String operations in OOP enable sophisticated text handling through inheritance-aware string representations, custom string-like classes, and polymorphic formatting systems. Exception hierarchies provide structured error handling that can be as specific or general as needed for your application.

7.5 Design Decision Framework

The choice between inheritance and composition depends on the relationship between objects. Use inheritance for "is-a" relationships where the derived class represents a specialized version of the base class. Use composition for "has-a" relationships where objects contain or use other objects as components.

Prefer composition over inheritance when possible because it provides more flexibility and avoids the complexity of deep inheritance hierarchies. Use inheritance when you need polymorphic behavior and have clear specialization relationships.

7.6 Real-World Applications

These advanced OOP concepts enable you to design professional software systems. Employee management systems use inheritance hierarchies to model different employee types while processing them polymorphically. Game development uses inheritance for character types and polymorphism for uniform processing. Scientific computing uses operator overloading to make mathematical objects work naturally with Python's syntax.

Exception hierarchies provide robust error handling in complex systems, allowing both specific error handling and general error management strategies. Duck typing enables plugin architectures and flexible integration of third-party components. String operations in inheritance enable sophisticated reporting systems, logging frameworks, and data presentation layers that maintain consistency while allowing customization.

7.7 Connection to Next Topics

With these advanced OOP concepts mastered, you are prepared for professional software development. Future topics will build on this foundation to explore design patterns, frameworks, and large-scale system architecture. The polymorphic thinking and inheritance design skills you have developed form the basis for understanding more advanced programming paradigms and software engineering practices.

Your progression from basic programming concepts through advanced object-oriented design represents a complete transformation in your ability to model complex systems and write maintainable, extensible code.