Tutorial by Dr. Ayes Chinmay
Date: 21/10/2025

Chapter 9: Classes and Magic Methods

Tutorial Overview

This comprehensive tutorial covers all aspects of Python classes and magic methods, from basic syntax to advanced features. Each section includes detailed explanations, code examples, and outputs to help you master object-oriented programming in Python.

9.1 Classes and Objects: Basic Syntax

Introduction to Classes

A **class** is a blueprint for creating objects. It defines attributes (data) and methods (functions) that objects of that class will have. Classes enable object-oriented programming (OOP) in Python.

Basic Class Definition

```
class Dog:
    """A simple Dog class"""

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

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

    def get_age(self):
        return f"{self.name} is {self.age} years old"

# Creating instances (objects)
dog1 = Dog("Buddy", 3)
dog2 = Dog("Max", 5)

# Using methods
print(dog1.bark())
print(dog2.get_age())
```

Output:

```
Buddy says Woof!
Max is 5 years old
```

Class Attributes vs Instance Attributes

```
class Circle:
    # Class attribute (shared by all instances)
    pi = 3.14159

    def __init__(self, radius):
        # Instance attribute (unique to each instance)
        self.radius = radius

    def area(self):
        return Circle.pi * self.radius ** 2

circle1 = Circle(5)
circle2 = Circle(10)

print(f"Circle 1 area: {circle1.area():.2f}")
print(f"Circle 2 area: {circle2.area():.2f}")
print(f"Pi value: {Circle.pi}")
```

Output:

```
Circle 1 area: 78.54
Circle 2 area: 314.16
Pi value: 3.14159
```

9.2 More About Instance Variables

Dynamic Instance Variables

Instance variables can be added to objects dynamically, even after the object is created.

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

person = Person("Alice")
print(f"Name: {person.name}")

# Adding new instance variable dynamically
person.age = 30
person.city = "New York"

print(f"Age: {person.age}")
print(f"City: {person.city}")
```

Output:

```
Name: Alice
Age: 30
City: New York
```

Instance Variables in Methods

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

    def deposit(self, amount):
        self.balance += amount
        self.transactions.append(f"Deposit: ${amount}")
        return f"Deposited ${amount}. New balance: ${self.balance}"

    def withdraw(self, amount):
        if amount &lt;= self.balance:
            self.balance -= amount
            self.transactions.append(f"Withdrawal: ${amount}")
            return f"Withdrew ${amount}. New balance: ${self.balance}"
        return "Insufficient funds"

    def get_statement(self):
        print(f"Account holder: {self.owner}")
        print(f"Current balance: ${self.balance}")
        print("Transactions:")
        for transaction in self.transactions:
            print(f"  - {transaction}")

account = BankAccount("John Doe", 1000)
print(account.deposit(500))
print(account.withdraw(200))
account.get_statement()
```

Output:

```
Deposited $500. New balance: $1500
Withdrew $200. New balance: $1300
Account holder: John Doe
Current balance: $1300
Transactions:
  - Deposit: $500
  - Withdrawal: $200
```

9.3 The `__init__` and `__new__` Methods

The `__init__` Method (Initializer)

The `__init__` method is called after an object is created to initialize its attributes.

```
class Book:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages
        print(f"Book created: {self.title}")

    def info(self):
        return f"{self.title} by {self.author} ({self.pages} pages)"

book1 = Book("Python Programming", "John Smith", 450)
book2 = Book("Data Science", "Jane Doe", 380)
print(book1.info())
print(book2.info())
```

Output:

```
Book created: Python Programming
Book created: Data Science
Python Programming by John Smith (450 pages)
Data Science by Jane Doe (380 pages)
```

The `__new__` Method (Constructor)

The `__new__` method is called before `__init__` and is responsible for creating the object instance.

```
class Singleton:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            print("Creating new instance")
            cls._instance = super().__new__(cls)
        else:
            print("Returning existing instance")
        return cls._instance

    def __init__(self):
        print("Initializing instance")

# Creating instances
s1 = Singleton()
print(f"Instance 1: {id(s1)}")

s2 = Singleton()
print(f"Instance 2: {id(s2)}")
print(f"Same instance? {s1 is s2}")
```

Output:

```
Creating new instance
Initializing instance
Instance 1: 140123456789012
Returning existing instance
Initializing instance
Instance 2: 140123456789012
Same instance? True
```

9.4 Classes and the Forward Reference Problem

Understanding Forward References

When a class references itself or another class not yet defined, you need to use forward references.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None  # Forward reference to another Node

    def set_next(self, node):
        self.next = node

    def display_chain(self):
        current = self
        result = []
        while current is not None:
            result.append(str(current.data))
            current = current.next
        return " -&gt; ".join(result)

# Creating a linked list
node1 = Node(10)
node2 = Node(20)
node3 = Node(30)

node1.set_next(node2)
node2.set_next(node3)

print("Linked List:")
print(node1.display_chain())
```

Output:

```
Linked List:
```

```
10 -&gt; 20 -&gt; 30
```

Using Type Hints with Forward References

```
from __future__ import annotations

class TreeNode:
    def __init__(self, value: int):
        self.value = value
        self.left: TreeNode | None = None
        self.right: TreeNode | None = None

    def insert(self, value: int) -&gt; None:
        if value &lt; self.value:
            if self.left is None:
                self.left = TreeNode(value)
            else:
                self.left.insert(value)
        else:
            if self.right is None:
                self.right = TreeNode(value)
            else:
                self.right.insert(value)

    def inorder(self) -&gt; list[int]:
        result = []
        if self.left:
            result.extend(self.left.inorder())
        result.append(self.value)
        if self.right:
            result.extend(self.right.inorder())
        return result

root = TreeNode(50)
root.insert(30)
root.insert(70)
root.insert(20)
root.insert(40)

print("Inorder traversal:", root.inorder())
```

Output:

```
Inorder traversal: [20, 30, 40, 50, 70]
```

9.5 Methods Generally

Instance Methods

Instance methods take `self` as the first parameter and can access instance attributes.

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

    def area(self):
        """Instance method to calculate area"""
        return self.width * self.height

    def perimeter(self):
        """Instance method to calculate perimeter"""
        return 2 * (self.width + self.height)

    def resize(self, factor):
        """Instance method to resize rectangle"""
        self.width *= factor
        self.height *= factor

rect = Rectangle(5, 3)
print(f"Area: {rect.area()}")
print(f"Perimeter: {rect.perimeter()}")

rect.resize(2)
print(f"After resize - Area: {rect.area()}")
```

Output:

```
Area: 15
Perimeter: 16
After resize - Area: 60
```

Class Methods

Class methods use `@classmethod` decorator and take `cls` as the first parameter.

```
class Employee:
    company_name = "Tech Corp"
    employee_count = 0

    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.employee_count += 1

    @classmethod
    def get_employee_count(cls):
        return f"{cls.company_name} has {cls.employee_count} employees"

    @classmethod
    def from_string(cls, emp_string):
        """Alternative constructor"""
        name, salary = emp_string.split(',')
        return cls(name, int(salary))

emp1 = Employee("Alice", 50000)
emp2 = Employee("Bob", 60000)
emp3 = Employee.from_string("Charlie,55000")

print(Employee.get_employee_count())
print(f"{emp3.name} earns ${emp3.salary}")
```

Output:

```
Tech Corp has 3 employees
Charlie earns $55000
```

Static Methods

Static methods use `@staticmethod` decorator and don't take `self` or `cls` parameters.

```
class MathOperations:
    @staticmethod
    def add(x, y):
        return x + y

    @staticmethod
    def multiply(x, y):
        return x * y

    @staticmethod
    def is_prime(n):
        if n &lt; 2:
            return False
        for i in range(2, int(n ** 0.5) + 1):
            if n % i == 0:
                return False
        return True

print(f"5 + 3 = {MathOperations.add(5, 3)}")
print(f"5 * 3 = {MathOperations.multiply(5, 3)}")
print(f"Is 17 prime? {MathOperations.is_prime(17)}")
print(f"Is 20 prime? {MathOperations.is_prime(20)}")
```

Output:

```
5 + 3 = 8
5 * 3 = 15
```

```
Is 17 prime? True
Is 20 prime? False
```

9.6 Public and Private Variables and Methods

Public vs Private Members

In Python, private members are indicated by a leading underscore (convention) or double underscore (name mangling).

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

    def get_balance(self):
        """Public method to access private data"""
        return self._balance

    def __validate_pin(self, pin):
        """Private method"""
        return pin == self.__pin

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

account = BankAccount("12345", 1000)
print(f"Balance: ${account.get_balance()}")
print(account.withdraw(200, "1234"))
print(account.withdraw(200, "wrong"))
```

Output:

```
Balance: $1000
Withdrew $200. New balance: $800
Invalid PIN
```

Property Decorators

```
class Temperature:
    def __init__(self, celsius):
        self._celsius = celsius
    @property
    def celsius(self):
        """Getter for celsius"""
        return self._celsius

    @celsius.setter
    def celsius(self, value):
        """Setter for celsius"""
        if value &lt; -273.15:
            raise ValueError("Temperature below absolute zero!")
        self._celsius = value

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

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

temp = Temperature(25)
print(f"{temp.celsius}°C = {temp.fahrenheit}°F")

temp.fahrenheit = 100
print(f"{temp.celsius:.1f}°C = {temp.fahrenheit}°F")
```

Output:

```
25°C = 77.0°F
37.8°C = 100.0°F
```

9.7 Inheritance

Basic Inheritance

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

    def make_sound(self):
        return "Some generic sound"

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

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name, "Dog")
        self.breed = breed
    def make_sound(self):
        return "Woof! Woof!"

    def fetch(self):
        return f"{self.name} is fetching the ball!"

class Cat(Animal):
    def __init__(self, name, color):
        super().__init__(name, "Cat")
        self.color = color

    def make_sound(self):
        return "Meow!"

dog = Dog("Buddy", "Golden Retriever")
cat = Cat("Whiskers", "Orange")

print(dog.info())
print(dog.make_sound())
print(dog.fetch())
print()
print(cat.info())
print(cat.make_sound())
```

Output:

```
Buddy is a Dog
Woof! Woof!
Buddy is fetching the ball!

Whiskers is a Cat
Meow!
```

Inheritance with Method Overriding

```
class Shape:
    def __init__(self, color):
        self.color = color

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

    def describe(self):
        return f"A {self.color} shape with area {self.area()}"

class Square(Shape):
    def __init__(self, color, side):
        super().__init__(color)
        self.side = side

    def area(self):
        return self.side ** 2
```

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

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

square = Square("red", 5)
circle = Circle("blue", 3)

print(square.describe())
print(circle.describe())
```

Output:

```
A red shape with area 25
A blue shape with area 28.27431
```

9.8 Multiple Inheritance

Basic Multiple Inheritance

```
class Flyer:
    def fly(self):
        return "I can fly!"

class Swimmer:
    def swim(self):
        return "I can swim!"

class Duck(Flyer, Swimmer):
    def __init__(self, name):
        self.name = name

    def quack(self):
        return f"{self.name} says: Quack!"

duck = Duck("Donald")
print(duck.quack())
print(duck.fly())
print(duck.swim())
```

Output:

```
Donald says: Quack!
I can fly!
I can swim!
```

Method Resolution Order (MRO)

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

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

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

class D(B, C):
    pass

d = D()
print(d.method())
print("MRO:", [cls.__name__ for cls in D.__mro__])
```

Output:

```
Method from B
MRO: ['D', 'B', 'C', 'A', 'object']
```

Diamond Problem Example

```
class Device:
    def __init__(self, brand):
        self.brand = brand
        print(f"Device.__init__ called with {brand}")

class Phone(Device):
    def __init__(self, brand, phone_model):
        super().__init__(brand)
        self.phone_model = phone_model
        print(f"Phone.__init__ called")

class Camera(Device):
    def __init__(self, brand, megapixels):
        super().__init__(brand)
        self.megapixels = megapixels
        print(f"Camera.__init__ called")

class Smartphone(Phone, Camera):
    def __init__(self, brand, phone_model, megapixels):
        Phone.__init__(self, brand, phone_model)
        self.megapixels = megapixels
        print(f"Smartphone.__init__ called")

    def info(self):
        return f"{self.brand} {self.phone_model} with {self.megapixels}MP camera"

phone = Smartphone("Apple", "iPhone 15", 48)
print(phone.info())
```

Output:

```
Device.__init__ called with Apple
Phone.__init__ called
Smartphone.__init__ called
Apple iPhone 15 with 48MP camera
```

9.9 Magic Methods, Summarized

Common Magic Methods Overview

Magic methods (also called dunder methods) allow you to customize the behavior of your classes.

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

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

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

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

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __len__(self):
        return int((self.x**2 + self.y**2)**0.5)

v1 = Vector(3, 4)
v2 = Vector(1, 2)
v3 = v1 + v2

print(f"v1: {v1}")
print(f"v2: {v2}")
print(f"v1 + v2: {v3}")
print(f"v1 == v2: {v1 == v2}")
print(f"Length of v1: {len(v1)}")
```

Output:

```
v1: Vector(3, 4)
v2: Vector(1, 2)
v1 + v2: Vector(4, 6)
v1 == v2: False
Length of v1: 5
```

9.10 Magic Methods in Detail

9.10.1 String Representation in Python Classes

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

    def __str__(self):
        """User-friendly string representation"""
        return f"{self.name} - ${self.price} (Stock: {self.quantity})"

    def __repr__(self):
        """Developer-friendly representation"""
        return f"Product(name='{self.name}', price={self.price}, quantity={self.quantity}

product = Product("Laptop", 999.99, 5)
print(str(product))
print(repr(product))
print(f"Product list: {[product]}")
```

Output:

```
Laptop - $999.99 (Stock: 5)
Product(name='Laptop', price=999.99, quantity=5)
Product list: [Product(name='Laptop', price=999.99, quantity=5)]
```

9.10.2 The Object Representation Methods

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

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

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

    def __format__(self, format_spec):
        if format_spec == 'polar':
            r = (self.x**2 + self.y**2)**0.5
            theta = __import__('math').atan2(self.y, self.x)
            return f"(r={r:.2f}, θ={theta:.2f})"
        return str(self)

p = Point(3, 4)
print(f"str: {str(p)}")
print(f"repr: {repr(p)}")
print(f"polar: {p:polar}")
```

Output:

```
str: (3, 4)
repr: Point(3, 4)
polar: (r=5.00, θ=0.93)
```

9.10.3 Comparison Methods

```
class Student:
    def __init__(self, name, grade):
        self.name = name
        self.grade = grade

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

    def __lt__(self, other):
        return self.grade &lt; other.grade

    def __le__(self, other):
        return self.grade &lt;= other.grade

    def __gt__(self, other):
        return self.grade &gt; other.grade

    def __ge__(self, other):
        return self.grade &gt;= other.grade

    def __ne__(self, other):
        return self.grade != other.grade

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

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

print("Sorted students:")
```

```
for student in sorted(students):
    print(f"  {student}")

print(f"\nBob == Diana: {students[1] == students[3]}")
print(f"Bob &gt; Alice: {students[1] &gt; students[0]}")
```

Output:

```
Sorted students:
  Student('Charlie', 78)
  Student('Alice', 85)
  Student('Bob', 92)
  Student('Diana', 92)

Bob == Diana: True
Bob &gt; Alice: True
```

9.10.4 Arithmetic Operator Methods

```
class Complex:
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    def __add__(self, other):
        return Complex(self.real + other.real, self.imag + other.imag)

    def __sub__(self, other):
        return Complex(self.real - other.real, self.imag - other.imag)

    def __mul__(self, other):
        real = self.real * other.real - self.imag * other.imag
        imag = self.real * other.imag + self.imag * other.real
        return Complex(real, imag)

    def __truediv__(self, other):
        denominator = other.real**2 + other.imag**2
        real = (self.real * other.real + self.imag * other.imag) / denominator
        imag = (self.imag * other.real - self.real * other.imag) / denominator
        return Complex(real, imag)

    def __str__(self):
        sign = '+' if self.imag &gt;= 0 else '-'
        return f"{self.real}{sign}{abs(self.imag)}i"

c1 = Complex(3, 4)
c2 = Complex(1, 2)

print(f"c1 = {c1}")
print(f"c2 = {c2}")
print(f"c1 + c2 = {c1 + c2}")
print(f"c1 - c2 = {c1 - c2}")
print(f"c1 * c2 = {c1 * c2}")
print(f"c1 / c2 = {c1 / c2}")
```

Output:

```
c1 = 3+4i
c2 = 1+2i
c1 + c2 = 4+6i
c1 - c2 = 2+2i
c1 * c2 = -5+10i
c1 / c2 = 2.2-0.4i
```

9.10.5 Unary Arithmetic Methods

```
class Number:
    def __init__(self, value):
        self.value = value

    def __neg__(self):
        return Number(-self.value)

    def __pos__(self):
        return Number(+self.value)

    def __abs__(self):
        return Number(abs(self.value))

    def __invert__(self):
        return Number(~self.value)

    def __str__(self):
        return str(self.value)

n = Number(-5)
print(f"Original: {n}")
print(f"Negative: {-n}")
print(f"Positive: {+n}")
print(f"Absolute: {abs(n)}")

n2 = Number(10)
print(f"\nNumber: {n2}")
print(f"Inverted: {~n2}")
```

Output:

```
Original: -5
Negative: 5
Positive: -5
Absolute: 5
```

```
Number: 10
Inverted: -11
```

9.10.6 Reflection (Reverse-Order) Methods

```
class MyNumber:
    def __init__(self, value):
        self.value = value

    def __add__(self, other):
        print(f"__add__ called: {self.value} + {other}")
        return MyNumber(self.value + other)

    def __radd__(self, other):
        print(f"__radd__ called: {other} + {self.value}")
        return MyNumber(other + self.value)

    def __mul__(self, other):
        print(f"__mul__ called: {self.value} * {other}")
        return MyNumber(self.value * other)

    def __rmul__(self, other):
        print(f"__rmul__ called: {other} * {self.value}")
        return MyNumber(other * self.value)

    def __str__(self):
        return str(self.value)

n = MyNumber(5)

result1 = n + 10
print(f"Result: {result1}\n")

result2 = 10 + n
print(f"Result: {result2}\n")

result3 = n * 3
print(f"Result: {result3}\n")

result4 = 3 * n
print(f"Result: {result4}")
```

Output:

```
__add__ called: 5 + 10
Result: 15

__radd__ called: 10 + 5
Result: 15

__mul__ called: 5 * 3
Result: 15
```

```
__rmul__ called: 3 * 5
Result: 15
```

9.10.7 In-Place Operator Methods

```python
class Counter:
    def __init__(self, value=0):
        self.value = value

    def __iadd__(self, other):
        print(f"__iadd__ called: {self.value} += {other}")
        self.value += other
        return self

    def __isub__(self, other):
        print(f"__isub__ called: {self.value} -= {other}")
        self.value -= other
        return self

    def __imul__(self, other):
        print(f"__imul__ called: {self.value} *= {other}")
        self.value *= other
        return self

    def __str__(self):
        return f"Counter({self.value})"

counter = Counter(10)
print(f"Initial: {counter}")

counter += 5
print(f"After +=: {counter}")

counter -= 3
print(f"After -=: {counter}")

counter *= 2
print(f"After *=: {counter}")
```

Output:

```
Initial: Counter(10)
__iadd__ called: 10 += 5
After +=: Counter(15)
__isub__ called: 15 -= 3
After -=: Counter(12)
__imul__ called: 12 *= 2
After *=: Counter(24)
```

9.10.8 Conversion Methods

```
class Fraction:
    def __init__(self, numerator, denominator):
        self.numerator = numerator
        self.denominator = denominator

    def __int__(self):
        return self.numerator // self.denominator

    def __float__(self):
        return self.numerator / self.denominator

    def __bool__(self):
        return self.numerator != 0

    def __str__(self):
        return f"{self.numerator}/{self.denominator}"

f1 = Fraction(7, 2)
f2 = Fraction(0, 5)

print(f"Fraction: {f1}")
print(f"As integer: {int(f1)}")
print(f"As float: {float(f1)}")
print(f"As boolean: {bool(f1)}")

print(f"\nFraction: {f2}")
print(f"As integer: {int(f2)}")
print(f"As float: {float(f2)}")
print(f"As boolean: {bool(f2)}")
```

Output:

```
Fraction: 7/2
As integer: 3
As float: 3.5
As boolean: True

Fraction: 0/5
As integer: 0
As float: 0.0
As boolean: False
```

9.10.9 Collection Class Methods

```
class MyList:
    def __init__(self, items=None):
        self.items = items if items else []

    def __len__(self):
        return len(self.items)
    def __getitem__(self, index):
        return self.items[index]

    def __setitem__(self, index, value):
        self.items[index] = value

    def __delitem__(self, index):
        del self.items[index]

    def __contains__(self, item):
        return item in self.items

    def __str__(self):
        return f"MyList({self.items})"

my_list = MyList([1, 2, 3, 4, 5])
print(f"List: {my_list}")
print(f"Length: {len(my_list)}")
print(f"Item at index 2: {my_list[2]}")

my_list[2] = 10
print(f"After modification: {my_list}")

print(f"10 in list: {10 in my_list}")
print(f"7 in list: {7 in my_list}")

del my_list[0]
print(f"After deletion: {my_list}")
```

Output:

```
List: MyList([1, 2, 3, 4, 5])
Length: 5
Item at index 2: 3
After modification: MyList([1, 2, 10, 4, 5])
10 in list: True
7 in list: False
After deletion: MyList([2, 10, 4, 5])
```

9.10.10 Implementing `__iter__` and `__next__`

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

    def __iter__(self):
        self.current = self.start
        return self

    def __next__(self):
        if self.current &lt;= 0:
            raise StopIteration
        self.current -= 1
        return self.current + 1

print("Countdown from 5:")
for num in Countdown(5):
    print(num, end=' ')

print("\n\nUsing iterator manually:")
countdown = Countdown(3)
iterator = iter(countdown)
print(next(iterator))
print(next(iterator))
print(next(iterator))
```

Output:

```
Countdown from 5:
5 4 3 2 1

Using iterator manually:
3
2
1
```

```
class FibonacciIterator:
    def __init__(self, max_count):
        self.max_count = max_count
        self.count = 0
        self.a, self.b = 0, 1

    def __iter__(self):
        return self

    def __next__(self):
        if self.count &gt;= self.max_count:
            raise StopIteration
        self.count += 1
        result = self.a
        self.a, self.b = self.b, self.a + self.b
        return result

print("First 10 Fibonacci numbers:")
for num in FibonacciIterator(10):
    print(num, end=' ')
```

Output:

```
First 10 Fibonacci numbers:
0 1 1 2 3 5 8 13 21 34
```

9.11 Supporting Multiple Argument Types

Type Checking and Multiple Argument Types

```
class Vector:
    def __init__(self, x, y=None):
        if isinstance(x, (list, tuple)) and y is None:
            self.x, self.y = x[0], x[1]
        elif isinstance(x, Vector) and y is None:
            self.x, self.y = x.x, x.y
        elif y is not None:
            self.x, self.y = x, y
        else:
            raise TypeError("Invalid arguments")

    def __add__(self, other):
        if isinstance(other, Vector):
            return Vector(self.x + other.x, self.y + other.y)
        elif isinstance(other, (int, float)):
            return Vector(self.x + other, self.y + other)
        else:
            raise TypeError(f"Cannot add Vector and {type(other)}")

    def __mul__(self, other):
        if isinstance(other, (int, float)):
            return Vector(self.x * other, self.y * other)
        elif isinstance(other, Vector):
            # Dot product
            return self.x * other.x + self.y * other.y
        else:
            raise TypeError(f"Cannot multiply Vector and {type(other)}")

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

# Different ways to create vectors
v1 = Vector(3, 4)
v2 = Vector([5, 6])
v3 = Vector(v1)

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

# Operations with different types
print(f"\nv1 + v2: {v1 + v2}")
print(f"v1 + 10: {v1 + 10}")
print(f"v1 * 2: {v1 * 2}")
print(f"v1 * v2 (dot product): {v1 * v2}")
```

Output:

```
v1: Vector(3, 4)
v2: Vector(5, 6)
```

```
v3: Vector(3, 4)

v1 + v2: Vector(8, 10)
v1 + 10: Vector(13, 14)
v1 * 2: Vector(6, 8)
v1 * v2 (dot product): 39
```

Using `isinstance` for Type Checking

```
class Calculator:
    @staticmethod
    def calculate(a, b, operation='+'):
        # Type checking and conversion
        if not isinstance(a, (int, float)):
            try:
                a = float(a)
            except ValueError:
                return "Error: Invalid first operand"

        if not isinstance(b, (int, float)):
            try:
                b = float(b)
            except ValueError:
                return "Error: Invalid second operand"

        operations = {
            '+': lambda x, y: x + y,
            '-': lambda x, y: x - y,
            '*': lambda x, y: x * y,
            '/': lambda x, y: x / y if y != 0 else "Error: Division by zero"
        }

        if operation not in operations:
            return "Error: Invalid operation"

        return operations[operation](a, b)

print(Calculator.calculate(10, 5, '+'))
print(Calculator.calculate("15", 3, '*'))
print(Calculator.calculate(20, "4", '/'))
print(Calculator.calculate(10, 0, '/'))
```

Output:

```
15
45.0
5.0
Error: Division by zero
```

9.12 Setting and Getting Attributes Dynamically

Using `getattr` and `setattr`

```
class DynamicObject:
    def __init__(self):
        self.name = "Dynamic"
        self.value = 100

obj = DynamicObject()

# Getting attributes dynamically
print("Using getattr:")
print(f"name: {getattr(obj, 'name')}")
print(f"value: {getattr(obj, 'value')}")
print(f"missing (with default): {getattr(obj, 'missing', 'Not Found')}")

# Setting attributes dynamically
print("\nUsing setattr:")
setattr(obj, 'new_attribute', 'Hello')
setattr(obj, 'number', 42)

print(f"new_attribute: {obj.new_attribute}")
print(f"number: {obj.number}")

# Checking if attribute exists
print("\nUsing hasattr:")
print(f"Has 'name': {hasattr(obj, 'name')}")
print(f"Has 'missing': {hasattr(obj, 'missing')}")

# Deleting attributes
print("\nUsing delattr:")
delattr(obj, 'value')
print(f"Has 'value' after deletion: {hasattr(obj, 'value')}")
```

Output:

```
Using getattr:
name: Dynamic
value: 100
missing (with default): Not Found

Using setattr:
new_attribute: Hello
number: 42

Using hasattr:
Has 'name': True
Has 'missing': False

Using delattr:
Has 'value' after deletion: False
```

Implementing `__getattr__` and `__setattr__`

```
class SmartDict:
    def __init__(self):
        self.__dict__['_data'] = {}

    def __getattr__(self, name):
        print(f"__getattr__ called for '{name}'")
        if name in self._data:
            return self._data[name]
        raise AttributeError(f"'{type(self).__name__}' has no attribute '{name}'")

    def __setattr__(self, name, value):
        print(f"__setattr__ called: {name} = {value}")
        if name.startswith('_'):
            self.__dict__[name] = value
        else:
            self.__dict__['_data'][name] = value

    def __delattr__(self, name):
        print(f"__delattr__ called for '{name}'")
        if name in self._data:
            del self._data[name]
        else:
            raise AttributeError(f"'{type(self).__name__}' has no attribute '{name}'")

    def keys(self):
        return self._data.keys()

obj = SmartDict()
obj.x = 10
obj.y = 20

print(f"\nAccessing x: {obj.x}")
print(f"Accessing y: {obj.y}")

print(f"\nAll keys: {list(obj.keys())}")

del obj.x
print(f"Keys after deletion: {list(obj.keys())}")
```

Output:

```
__setattr__ called: x = 10
__setattr__ called: y = 20

__getattr__ called for 'x'
Accessing x: 10
__getattr__ called for 'y'
Accessing y: 20

All keys: ['x', 'y']
```

```
__delattr__ called for 'x'
Keys after deletion: ['y']
```

Property-Based Dynamic Attributes

```
class Configuration:
    def __init__(self):
        self._config = {
            'debug': False,
            'timeout': 30,
            'max_retries': 3
        }

    def __getattr__(self, name):
        if name in self._config:
            return self._config[name]
        raise AttributeError(f"Configuration has no option '{name}'")

    def __setattr__(self, name, value):
        if name == '_config':
            super().__setattr__(name, value)
        elif name in self.__dict__.get('_config', {}):
            self._config[name] = value
        else:
            raise AttributeError(f"Cannot set unknown configuration option '{name}'")

    def add_option(self, name, default_value):
        self._config[name] = default_value

    def show_all(self):
        print("Configuration:")
        for key, value in self._config.items():
            print(f"  {key}: {value}")

config = Configuration()
print("Initial configuration:")
config.show_all()

print(f"\nAccessing debug: {config.debug}")
print(f"Accessing timeout: {config.timeout}")

config.debug = True
config.timeout = 60

print("\nAfter modifications:")
config.show_all()

config.add_option('verbose', True)
print("\nAfter adding option:")
config.show_all()
print(f"Verbose: {config.verbose}")
```

Output:

```
Initial configuration:
Configuration:
  debug: False
  timeout: 30
  max_retries: 3

Accessing debug: False
Accessing timeout: 30

After modifications:
Configuration:
  debug: True
  timeout: 60
  max_retries: 3

After adding option:
Configuration:
  debug: True
  timeout: 60
  max_retries: 3
  verbose: True
Verbose: True
```

Chapter Summary

Key Concepts Covered

1. **Classes and Objects**: Basic syntax for defining classes and creating objects
2. **Instance Variables**: Managing object state through instance attributes
3. **Initialization Methods**: Using `__init__` and `__new__` for object creation
4. **Forward References**: Handling self-referencing and circular dependencies
5. **Methods**: Instance methods, class methods, and static methods
6. **Encapsulation**: Public, protected, and private members
7. **Inheritance**: Creating class hierarchies and method overriding
8. **Multiple Inheritance**: Working with multiple parent classes and MRO
9. **Magic Methods**: Customizing object behavior through special methods
10. **Operator Overloading**: Implementing arithmetic and comparison operators
11. **Type Flexibility**: Supporting multiple argument types
12. **Dynamic Attributes**: Runtime attribute manipulation

Best Practices

- Use clear, descriptive class and method names
- Follow Python naming conventions (PEP 8)
- Implement `__str__` and `__repr__` for better debugging
- Use properties for controlled attribute access
- Prefer composition over inheritance when appropriate
- Document your classes with docstrings
- Use type hints for better code clarity
- Implement magic methods to make classes more Pythonic

Common Pitfalls to Avoid

- Forgetting to call `super().__init__()` in derived classes
- Modifying class attributes when you meant to modify instance attributes
- Not understanding method resolution order in multiple inheritance
- Implementing incomplete magic method sets (e.g., comparison methods)
- Overusing private attributes (leading underscores)
- Creating circular dependencies in class relationships

Practice Exercises

Exercise 1: Create a Library Management System

Implement classes for `Book`, `Member`, and `Library` with appropriate attributes and methods.

Exercise 2: Design a Shape Hierarchy

Create an abstract `Shape` base class and derive `Circle`, `Rectangle`, and `Triangle` classes with area and perimeter calculations.

Exercise 3: Build a Custom Container

Implement a `Stack` class with magic methods for `len`, indexing, and iteration.

Exercise 4: Develop a Banking System

Create `Account`, `SavingsAccount`, and `CheckingAccount` classes with transaction history and interest calculations.

Exercise 5: Implement a Vector Mathematics Library

Build a comprehensive `Vector` class with support for vector operations, dot product, cross product, and magnitude calculations.

Conclusion

This tutorial has covered all major aspects of Python classes and magic methods from sections 9.1 through 9.12. Understanding these concepts is fundamental to writing effective object-oriented Python code. Practice implementing your own classes and experimenting with magic methods to fully master these concepts.

Remember that object-oriented programming is about modeling real-world entities and their relationships in code. Use classes to create clean, maintainable, and reusable code structures in your projects.