# 3.1 编写面向对象的程序，练习类的定义和使用

## 课程目标

- 了解类与对象的概念
- 掌握类的定义与使用方法
- 熟练使用类的属性和方法
- 掌握继承与多态的概念

## 课程内容

1. 类与对象的定义
2. 类的属性与方法
3. 类的实例化与调用
4. 继承与多态的概念
5. 实践练习：编写面向对象的程序

## 实践指导

1. 编写一个简单的类，定义属性和方法
2. 实例化类对象，调用类的方法
3. 继承一个已有的类，重写父类的方法
4. 实现多态的效果，调用不同子类的同名方法

## 参考资料

- Python 面向对象编程
- Python 面向对象编程

## 练习题 1

1. 编写一个动物类，包含属性：种类、年龄、性别，方法：吃、睡
2. 编写一个狗类，继承动物类，包含方法：叫
3. 实例化动物和狗对象，调用各自的方法
4. 实现多态的效果，调用动物和狗对象的同名方法

## 代码实例

```
class Animal:
    def __init__(self
                 , species: str
                 , age: int
```

```
    def eat(self):
        print(f"{self.species} is eating.")

    def sleep(self):
        print(f"{self.species} is sleeping.")

class Dog(Animal):
    def bark(self):
        print(f"{self.species} is barking.")

if __name__ == "__main__":
    animal
    dog = Dog
    animal.eat()
    animal.sleep()
    dog.eat()
    dog.sleep()
    dog.bark()
```

## 练习题 2

1. 编写一个图形类，包含属性：颜色、形状，方法：绘制
2. 编写一个矩形类，继承图形类，包含属性：长、宽，方法：计算面积
3. 实例化图形和矩形对象，调用各自的方法
4. 实现多态的效果，调用图形和矩形对象的同名方法

## 代码实例

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

    def draw(self):
        print(f"Drawing a {self.color} {self.shape}.")

class Rectangle(Shape):
    def __init__(self, color: str, shape: str, length: int, width: int):
        super().__init__(color, shape)
        self.length = length
        self.width = width

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

if __name__ == "__main__":
    shape = Shape("red", "circle")
    rectangle = Rectangle("blue", "rectangle", 5, 3)
    shape.draw()
    rectangle.draw()
```

```
    print(f"The area of the rectangle is {rectangle.area()}.")
```

## 练习题 3

1. 编写一个人类，包含属性：姓名、年龄、性别，方法：吃、睡
2. 编写一个学生类，继承人类，包含属性：学号、班级，方法：学习
3. 编写一个老师类，继承人类，包含属性：工号、科目，方法：教学
4. 实例化人、学生和老师对象，调用各自的方法
5. 实现多态的效果，调用人、学生和老师对象的同名方法

## 代码实例

```
class Person:
    def __init__(self, name: str, age: int

    def eat(self):
        print(f"{self.name} is eating.")

    def sleep(self):
        print(f"{self.name} is sleeping.")

class Student(Person):
    def __init__(self, name: str, age: int, student_id: str, class_name: str):
        super().__init__(name, age)
        self.student_id = student_id
        self.class_name = class_name

    def study(self):
        print(f"{self.name} is studying.")

class Teacher(Person):
    def __init__(self, name: str, age: int, teacher_id: str, subject: str):
        super().__init__(name, age)
        self.teacher_id = teacher_id
        self.subject = subject

    def teach(self):
        print(f"{self.name} is teaching.")

if __name__ == "__main__":
    person = Person("Alice", 25)
    student = Student("Bob", 20, "2021001", "Class A")
    teacher = Teacher("Charlie", 30, "T2021001", "Math")
    person.eat()
    person.sleep()
    student.eat()
```

```
    student.sleep()
    student.study()
    teacher.eat()
    teacher.sleep()
    teacher.teach()
```

### 面向对象编程作业题

**任务描述：**

设计一个简单的图书管理系统，要求如下：

1. **图书类（Book）**：
   - 属性：书名（title）、作者（author）、ISBN号（isbn）、价格（price）
   - 方法：
     - `__init__`：初始化图书信息
     - `display_info`：打印图书的详细信息
2. **图书馆类（Library）**：
   - 属性：图书集合（一个包含多本图书的列表）
   - 方法：
     - `__init__`：初始化图书馆，开始时图书集合为空
     - `add_book`：向图书馆添加一本图书
     - `remove_book_by_isbn`：通过ISBN号移除一本图书
     - `display_all_books`：打印图书馆中所有图书的信息

**要求：**

- 使用面向对象的方式实现上述需求。
- 对图书进行添加、移除操作后，能够正确显示图书馆中的图书信息。

**附加挑战（可选）：**

- 实现一个搜索功能，通过书名或作者搜索图书，并显示相关图书的信息。
- 对图书按价格进行排序并显示。

**提示：**

- 可以使用列表来存储图书馆中的图书对象。
- 在`Library`类中，遍历图书列表来实现显示和移除图书的功能。

`3.2 编写高级数据结构的练习题，使用生成器和装饰器优化代码`

# 3.2 编写高级数据结构的练习题，使用生成器和装饰器优化代码

## 练习题 1：实现一个生成器版本的链表

**题目描述：**
实现一个链表数据结构，并使用生成器遍历链表中的元素。

**要求：**

1. 定义一个 `Node` 类，表示链表的节点。
2. 定义一个 `LinkedList` 类，表示链表。
3. 在 `LinkedList` 类中实现一个生成器方法 `__iter__`，用于遍历链表中的元素。

**示例代码：**

```
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def append(self, value):
        new_node = Node(value)
        if not self.head:
            self.head = new_node
            return
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node

    def __iter__(self):
        current = self.head
        while current:
            yield current.value
            current = current.next

# 测试代码
linked_list = LinkedList()
linked_list.append(1)
linked_list.append(2)
linked_list.append(3)

for value in linked_list:
    print(value)  # 输出：1 2 3
```

## 练习题 2：实现一个装饰器优化的缓存机制

**题目描述：**
实现一个装饰器，用于缓存函数的计算结果，以提高函数的执行效率。

**要求：**

1. 定义一个 `cache` 装饰器，用于缓存函数的计算结果。
2. 使用该装饰器优化一个计算斐波那契数列的函数。

**示例代码：**

```
def cache(func):
    cached_results = {}

    def wrapper(*args):
        if args in cached_results:
            return cached_results[args]
        result = func(*args)
        cached_results[args] = result
        return result

    return wrapper

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

# 测试代码
print(fibonacci(10))  # 输出：55
print(fibonacci(20))  # 输出：6765
```

## 练习题 3：实现一个生成器版本的二叉树遍历

**题目描述：**
实现一个二叉树数据结构，并使用生成器实现中序遍历。

**要求：**

1. 定义一个 `TreeNode` 类，表示二叉树的节点。
2. 定义一个 `BinaryTree` 类，表示二叉树。
3. 在 `BinaryTree` 类中实现一个生成器方法 `inorder_traversal`，用于中序遍历二叉树。

**示例代码：**

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

class BinaryTree:
    def __init__(self, root_value):
        self.root = TreeNode(root_value)
```

```
    def insert(self, value):
        self._insert(self.root, value)

    def _insert(self, node, value):
        if value < node.value:
            if node.left is None:
                node.left = TreeNode(value)
            else:
                self._insert(node.left, value)
        else:
            if node.right is None:
                node.right = TreeNode(value)
            else:
                self._insert(node.right, value)

    def inorder_traversal(self):
        yield from self._inorder_traversal(self.root)

    def _inorder_traversal(self, node):
        if node:
            yield from self._inorder_traversal(node.left)
            yield node.value
            yield from self._inorder_traversal(node.right)

# 测试代码
binary_tree = BinaryTree(10)
binary_tree.insert(5)
binary_tree.insert(15)
binary_tree.insert(3)
binary_tree.insert(7)

for value in binary_tree.inorder_traversal():
    print(value)  # 输出: 3 5 7 10 15
```

## 练习题 4：实现一个装饰器优化的计时器

**题目描述：**
实现一个装饰器，用于计算函数的执行时间。

**要求：**

1. 定义一个 `timer` 装饰器，用于计算函数的执行时间。
2. 使用该装饰器优化一个执行时间较长的函数。

**示例代码：**

```
import time

def timer(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
```

```
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"Function {func.__name__} took {end_time - start_time:.4f} seconds")
        return result
    return wrapper

@timer
def slow_function():
    time.sleep(2)
    print("Function complete")

# 测试代码
slow_function()
# 输出:
# Function complete
# Function slow_function took 2.0001 seconds
```

## 练习题 5：实现一个生成器版本的优先级队列

**题目描述：**
实现一个优先级队列数据结构，并使用生成器遍历队列中的元素。

**要求：**

1. 定义一个 `PriorityQueue` 类，表示优先级队列。
2. 在 `PriorityQueue` 类中实现一个生成器方法 `__iter__`，用于遍历队列中的元素。

**示例代码：**

```
import heapq

class PriorityQueue:
    def __init__(self):
        self._queue = []
        self._index = 0

    def push(self, item, priority):
        heapq.heappush(self._queue, (-priority, self._index, item))
        self._index += 1

    def pop(self):
        return heapq.heappop(self._queue)[-1]

    def __iter__(self):
        while self._queue:
            yield self.pop()

# 测试代码
priority_queue = PriorityQueue()
priority_queue.push("task1", 1)
priority_queue.push("task2", 5)
```

```
priority_queue.push("task3", 3)

for task in priority_queue:
    print(task)  # 输出: task2 task3 task1
```

通过这些练习，将能够掌握如何使用生成器和装饰器优化高级数据结构的实现，提高代码的可读性和性能。