第六章 Python 类与面向对象编程

6.1 类和对象

6.1.1 面向对象编程简介

面向对象	OOP 通过**定义类**来模拟现实中的对象，将**数据（属性）和操作数据的方法（行为）**封装在一起。
类	Class 是一种蓝图或模板，定义了对象**共同拥有的属性和方法**。
对象	Object 是类的实例，拥有**具体的**状态（属性值）和行为（方法实现）。
主要特点	**封装**：将**数据和方法组合**，隐藏内部实现细节。 **继承**：允许**子类继承父类的属性和方法**，实现代码复用。 **多态**：**不同对象**对**同一方法**调用**作出不同响应**，提高程序灵活性。

6.1.2 Python 中的类和对象

类的定义	通过 `class` 关键字定义类。类可以包含属性（变量）和方法（函数） 命名规范：**驼峰首字母大写**
类属性	定义位置：**直接在类体中定义** 特点：所有实例共享的属性，公有。
实例属性	定义位置：**在构造方法 `__init__` 中**，通过 self 赋值 特点：每个对象拥有独立的数据，如 location、temperature、humidity
方法	定义位置：**在类内部定义**，与属性同级，**必须以 self 作为第一个参数** 特点：在调用时自动传入调用该方法的对象，方便访问和操作该对象的属性
实例	各自拥有独立的数据，共享类定义的行为和部分属性。

6.2 构造方法与析构方法

6.2.1 构造方法 `__init__`

作用	在对象创建时**自动调用**，初始化属性和状态
特点	支持多个参数、默认值和关键字参数，确保对象在创建时即处于有效状态。
示例	统计全国各地的站点数据，每个站点都具有的属性，例如位置、站点号等。

6.2.2 析构方法 `__del__`

作用	在对象销毁前执行清理工作
说明	由于 Python 有**自动垃圾回收机制**，通常无需手动调用析构方法。

```
    def __del__(self):
        print("天气预报对象正在被销毁")
forcast = Forecast('sunny')
del forcast #显式地删除对象（通常由垃圾回收机制自动处理）
```

```
class WeatherStation:
    #假设所有气象站使用同一个测量单位
    unit = 'metric'  #类属性
    # 构造方法：对象创建时自动调用，用于初始化
    def
__init__(self,location,temperture,humidity):
    #创建实例时需要提供的信息
        self.location = location
        #实例属性：每个气象站独有的地点，私有
        self.temperture = temperture
        self.humidity = humidity

    #实例方法：操作对象属性、定义行为、第一个参数必须为 self
    def display_info(self):
        print(f"{self.location}的当前天气状况：
         温度{self.temperture}°{self.unit},
         湿度{self.humidity}%")
ws_a = WeatherStation('北京',25,35)      #实例化对象
ws_b = WeatherStation('上海',30,40)
ws_a.specify = "custom"  #可以自己定义属性，动态加载属性
```

6.3 属性

6.3.1 实例属性与类属性

实例属性　绑定在具体对象上，每个气象站有自己独立的数据（如地点）。

类属性　绑定在类上，所有实例共享同一份数据（如 count）。

```
class StationCounter:
    count = 0   #类属性
    def __init__(self,location):
        self.location = location #实例属性
        StationCounter.count += 1
        #每创建一次，共用属性增加一

ws_a = StationCounter('北京')
ws_b = StationCounter('上海')
print(f'当前共有{StationCounter.count}个气象站')
StationCounter.count = 100  #直接修改类属性
ws_a.location = '广州'      #修改实例属性
print(f'当前共有{StationCounter.count}个气象站')
```

6.3.2 属性的访问与修改

访问　可以使用**对象.属性**或**类名.属性**。

修改　直接赋值可能在实例中创建一个同名属性，遮蔽类属性；建议通过类名修改类属性。

6.4 方法

6.4.1 实例方法

定义　在**类内部定义的函数**，**描述对象的行为**。

要求　① **第一个参数必须为 self**，代表调用该方法的对象实例

② 方法内部**通过 self 访问实例属性**和其他方法。

说明　1. 构造方法 `__init__`初始化传感器类型，将其保存在实例属性 `sensor_type` 中。

2. 实例方法 `read_data` 使用 `self.sensor_type` 读取数据并打印信息。

3. 调用 `temp_sensor.read_data()` 时，`temp_sensor` 会自动作为 `self` 传入方法中，输出对应数据。

```
# 实例方法
class Sensor:
    def __init__(self,sensor_type):
        self.sensor_type = sensor_type
    def read_data(self):
        print(f"{self.sensor_type}正在读取...")
temp_sensor = Sensor('温度传感器')
temp_sensor.read_data()
```

6.4.2 类方法

定义　① **使用 `@classmethod` 装饰器定义**，方法第一个参数通常为 `cls`，表示当前类。

② 用的不多，类方法可直接通过类名调用无需实例

用途　① 访问或修改类属性

② 备用构造方法（工厂方法）

说明　1. 类属性 `total_reports` 用于统计实例数量；构造方法中每创建一个实例即加 1。

2. `get_total_reports` 作为类方法可直接访问该类属性。

3. `create_from_string` 是工厂方法，根据预处理后的字符串创建新实例。

```
# 类方法
class WeatherReport:
    total_reports = 0
    def __init__(self,report):
        self.report = report
        WeatherReport.total_reports += 1
    @classmethod
    # 类方法，第一个参数是类本身，用 cls 表示
    def get_total_reports(cls):
        return cls.total_reports    #改变共有属性
    @classmethod
    def create_from_string(cls,report_str):
        report = report_str.strip(' ')
        return cls(report)  #返回
wr1 = WeatherReport('天气晴朗')   #通过实例创建
wr2 = WeatherReport.create_from_string('天气阴天')
#通过类创建，创建实例时可以调用
wr1.report
```

6.4.3 静态方法

定义　① **使用 `@staticmethod` 装饰器定义**，静态方法**不接收** self 或 cls 参数。

② 本质上是**定义在类内部的普通函数**

用途　① 适合做**小工具**：如数据格式转换、单位转换等。

② **代码组织**：将逻辑相关的工具函数归类，便于复用和管理。　③ 静态方法不依赖于类或实例的数据，仅对传入的参数执行运算。可以直接通过类名调用，提高代码的复用性和组织性。

6.5 封装、继承和多态

6.5.1 封装与数据隐藏

封装　将数据和相关方法封装在一起，隐藏内部实现。

规则　① **单下划线（_value）**：提示**受保护**，仅在类内部或子类中使用。

② **双下划线（__value）**：**私有**效果

6.5.2 继承

继承说明　允许子类继承父类的属性方法，扩展或重写父类行为

在子类的__init__方法中，通常使用 super().__init__(location)调用父类的构造函数，从而初始化父类定义的属性。

6.5.3 多态

多态　是面向对象编程中的一个重要特性，指的是不同的对象可以对同一方法调用做出不同的响应，而调用者无需关心这些对象的具体类型。换句话说，可以对一组不同的对象统一调用某个方法，而每个对象会根据自身的实现返回不同的结果。

这种机制使得代码更具有通用性和灵活性。

依赖　在 Python 中，多态主要依赖于**方法重写（Override）**

① **父类定义接口**：父类中定义一个方法，描述一种操作或行为。

② **子类重写方法**：各子类根据自己的特点，实现这个方法，提供具体的行为。

③ **调用统一接口**：外部代码只需知道父类的接口，不需要关心具体子类的实现细节。调用时，实际执行的是子类的重写方法。

```
# 多态
class WeatherForcast():
    def get_forcast(self):
        raise NotImplementedError("子类必须实现此方法")
class SunnyForcast(WeatherForcast):
    def get_forcast(self):
        return "晴天， enjoy the sunshine!"
class RainyForcast(WeatherForcast):
    def get_forcast(self):
        return "下雨， stay inside and keep hydrated."
def display_forcast(forcast:WeatherForcast):
    print(f'天气预报：{forcast.get_forcast()}')

forcast = SunnyForcast()
```

```
# 静态方法
class WeatherUtils: #适合做小工具
    @staticmethod
    def convert_c_to_f(celsius):    #摄氏度→华氏度
        return celsius * 9/5 + 32
    @staticmethod
    def convert_f_to_c(fahrenheit): #华氏度→摄氏度
        return (fahrenheit - 32) * 5/9
print(WeatherUtils.convert_c_to_f(25))
```

```
# 保护
class WeatherData:
    def __init__(self,temperature):
        #受保护属性，可以访问，但不建议
        self._raw_temperature = temperature
        #私有属性，无法访问
        self.__calibrated_temperature =
        temperature + 1
    #访问私有属性，需要使用实例方法
    def get_calibrated_temperature(self):
        #有更好的方式访问
        return self.__calibrated_temperature
data = WeatherData(25)    #实例对象
```

```
#继承
class WeatherStation:
    def __init__(self,location):
        self.location = location
    def display_info(self):
        print(f"{self.location}的当前天气状况")
#高级气象站类，增加传感信息
class AdvancedWeatherStation(WeatherStation):
    def __init__(self,location,sensors):
        #调用父类的构造方法__init__
        super().__init__(location)
        self.sensors = sensors
    def display_info(self):
        super().display_info()  #调用父类的方法
        print(f"传感器：{self.sensors}")
aws = AdvancedWeatherStation('北京',['温度'])
aws.display_info()  #调用子类的方法，overwrite
```

6.5.4 多继承与 MRO

多继承　一个类**可以同时继承多个父类**，当父类中存在同名方法时，Python 按照 **MRO（方法解析顺序）**查找。