python基础7_面向对象1

5.2. 搬家具拓展版本

一、面向对象：类和对象、魔法方法

1. 面向对象编程概述

1.1【了解】面向过程和面向对象的区别

- 面向过程
  - 把编程任务划分成一个一个的步骤，然后按照步骤分别去执行，适合开发**中小型项目**
  - 以 吃饭 举例：买菜、洗菜、煮饭、切菜.....
- 面向对象
  - 把构成问题事务分解成各个**对象**，适合**开发大型项目**
  - 以 吃饭 举例：找 饭馆 对象，饭馆提供菜和饭，不关心做菜和煮饭的内部过程

1.2【知道】类和对象介绍

- 类是对象的**模板**(不占内存空间)，对象是类的**实例**(占内存空间)。
- 类相当于**图纸**，对象相当于根据图纸制造的**实物**。

2. 面向对象编程

2.1【重点】定义类和方法

```
class 类名(object):
    def 方法名(self):
        pass
```

- 代码:

```
"""
# 1. 定义类，设计一个类型
格式:
#class 类名:
#class 类名(): # 前2个旧式写法，不推荐
class 类名(object):
    方法列表（不是真的是列表，只是多个函数的定义）

# object所有类的祖先
# 类名：大驼峰命名
"""
```

```
# 定义类:狗类 Dog
# class 类名(object):
#      多个方法


class Dog(object):
    def eat(self):
        print("啃骨头")

    def drink(self):
        print("喝水")
```

2.2【重点】创建对象和调用方法

- 创建对象：`对象变量名 = 类名()`
- 调用方法：`对象变量名.方法名()`

```
"""
1. 定义类，只是定义了一个类型
2. 根据类，创建对象（实例对象），  通过设计图创建一个实物
    格式：   实例对象变量名 = 类名()
3. 类里面方法如何调用
```

```
    对象变量.方法名字()
"""


# 1. 定义类，只是定义了一个类型
class Dog(object):
    def eat(self):
        print("啃骨头")

    def drink(self):
        print("喝水")


# 创建对象格式：实例对象变量名 = 类名()
dog1 = Dog()
# 对象变量.方法名字(), self不用处理
dog1.eat()
dog1.drink()
```

- 创建多个对象:

```
class Dog(object):
    def eat(self):
        print("啃骨头")
```

```
    def drink(self):
        print("喝水")


# 对象1 = 类名()
dog1 = Dog()
# 对象2 = 类名()
dog2 = Dog()
# 对象1
dog1.eat()
dog1.drink()
# 对象2
dog2.eat()
dog2.drink()
```

2.3【重点】添加和使用属性

定义/添加属性格式：

```
对象变量名.属性名 = 数据    # 第一次赋值是定义，第二次赋值是修改
```

- **首次赋值时会定义属性**，再次赋值改变属性

```
"""
给类添加属性:
1. 创建对象变量
2. 对象变量.属性 = 数值
    # 第一次赋值是添加属性，再次赋值是修改

如果使用属性
对象变量.属性
"""

class Dog(object):
    def eat(self):
        print("啃骨头")

    def drink(self):
        print("喝水")


# 1. 创建对象变量
dog1 = Dog()

# 2. 对象变量.属性 = 数值
dog1.age = 3    # 第一次赋值是定义

# 打印属性
print(dog1.age)
```

```
# 修改属性
dog1.age = 2
print(dog1.age)
```

2.5【记忆】self是什么

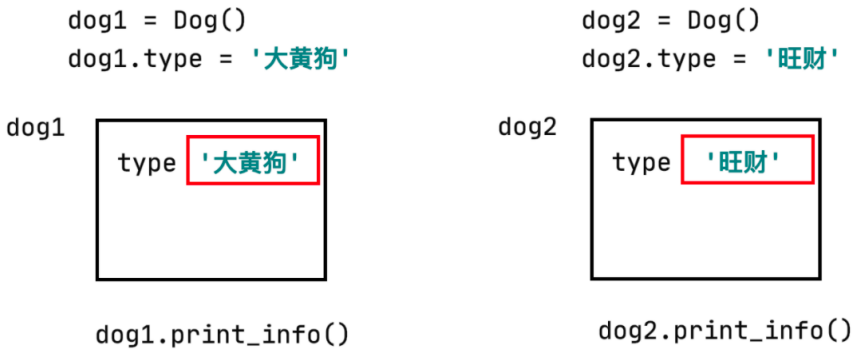
- 哪个对象调用方法，方法中self就是这个对象本身

```
"""
self是什么：哪个对象调用方法，方法中self就是这个对象
"""

class Dog(object):

    def print_info(self):
        print("测试代码：", id(self))


dog1 = Dog()
print("调用方法前：", id(dog1))

dog1.print_info()   # 解释器自动传递参数dog1，相当于print_info(dog1)
```

```
print("调用方法后：", id(dog1))

print("="*50)

dog2 = Dog()
print("调用方法前：", id(dog2))

dog2.print_info()    # 解释器自动传递参数dog2，相当于print_info(dog2)

print("调用方法后：", id(dog2))
```

2.5【理解】self的作用

- 为了**区分**不同对象，访问不同对象的属性和方法




```
def print_info(self):
    print(self.type)
```

1. dog1调用print_info时，self就是dog1，方法里面打印的是dog1.type，即为'大黄狗'
2. dog2调用print_info时，self就是dog2，方法里面打印的是dog2.type，即为'旺财'

```
"""
self作用：为了区分不同对象的属性和方法
"""


# 定义类
class Dog(object):
    # self作用：为了区分不同对象.(那个对象调用了方法,self就是这个对象本身)
    def print_info(self):
        print(self.name)


# 创建对象，实例化对象
dog1 = Dog()
# 添加属性
```

```
dog1.name = "大黄"
# 直接调用方法
# 解释器自动把dog1作为函数的第一个参数传递给self，相当于print_info(dog1)
dog1.print_info()

print("="*30)

dog2 = Dog()

dog2.name = "小黄"
# 解释器自动把dog2作为函数的第一个参数传递给self，相当于print_info(dog2)
dog2.print_info()
```

3. 魔法方法

3.1【重点】`__init__`方法

1. `__init__`方法的作用和特点
    - 作用：添加属性
    - 特点：创建对象的时候，实例化对象，自动调用`__init__`方法

```
"""
__init__方法:
    1. 作用:添加属性
    2. 特点:创建对象的时候,实例化对象,自动调用__init__方法
"""


class Dog(object):

    # 定义魔法方法__init__, 会在对象创建时被系统自动调用.
    def __init__(self):
        # 对象初始化方法, 给对象添加属性用的.
        self.type = "基多"
        print("对象被初始化了")

    def print_info(self):
        print(self.type)


# 1. 创建对象,实例化对象,系统会自动调用__init__方法,
#    系统使用对象dog1调用__init__方法,所以__init__中的self参数就是dog1
dog1 = Dog()
```

```
dog1.print_info()    # 相当于
print_info(dog1)
```

2. 不带参数和带参数的`__init__`方法的使用

```
# 不带参数
class 类名(object):
    def __init__(self):
        pass

# 实例化对象
对象名 = 类名()
#########################################
# 带参数
class 类名(object):
    def __init__(self, 形参1,  形参2 ……):
        pass

# 实例化对象
对象名 = 类名(实参1,  实参2, ……)
```

- 代码:

```
"""
__init__方法:
```

```
    1. 作用：添加属性
    2. 特点：创建对象的时候，实例化对象，自动调用__init__方法
    3. 设置参数，创建对象时，除了self参数不用人为处理，其它参数需要和__init__参数匹配
        对象名 = 类名(实参1， 实参2) ====》__init__(self, 形参1, 形参2)
"""


class Dog(object):

    # 带有参数的__init__方法, __init__除了self参数外,可以自定义参数
    def __init__(self, _type):
        self.type = _type
        print("对象初始化方法")

    def print_info(self):
        print(self.type)


# 创建对象，实例化对象，自动调用__init__方法
#    系统使用对象dog1调用__init__方法,并且传递参数dog1给self,"基多"给_type
dog1 = Dog("基多")
```

```
dog1.print_info()  # 相当于 print_info(dog1)

print("=" * 30)

#    系统使用对象dog2调用__init__方法,并且传递参数dog2给self,"拉布拉多"给_type
dog2 = Dog("拉布拉多")

dog2.print_info()  # 相当于 print_info(dog2)
```

3.2【重点】__str__方法

- __str__方法的作用：
  - __str__()方法作用主要返回对象属性信息，print(对象变量名)输出对象时直接输出__str__()方法返回的描述信息
  - __str__()方法的返回值必须是 **字符串类型**

```
"""
__str__方法：
    1. 返回值必须是字符串类型
    2. print(对象变量名)  对象变量名的位置替换为__str__()方法返回值的内容
"""
```

```python
class Dog(object):

    def __init__(self, _type, _age):
        # 对象初始化魔法方法，给对象添加属性使用的，会在创建对象时被系统自动调用
        self.type = _type
        self.age = _age
        print("对象初始化方法被调用了")

    def __str__(self):
        # 返回对象描述信息的魔法方法，
        # 当使用print(对象)时，系统会自动调用__str__方法获取对象描述，print打印对象描述信息
        # 格式化字符串
        # return "类型：%s 年龄：%d" %(self.type, self.age)
        return f"类型：{self.type} 年龄：{self.age}岁"


dog1 = Dog("基多", 2)

# <__main__.Dog object at 0x000001B81E6E5910>
```

```
# 类中没有实现__str__方法,返回对象地址信息
# print(dog1)

# 类型: 基多 年龄: 2
print(dog1)  # print(对象), 系统会自动打印对象.__str__() 返回的描述信息

print("对象dog1描述: ", dog1)
```

3.3【了解】`__del__`方法

- 对象销毁时会自动调用`__del__`方法

```
"""
在对象的生命周期结束(对象销毁)时, __del__()方法会自动被调用, 做一些清理工作
"""


class Dog(object):

    def __del__(self):
        print("对象被销毁了,做清理动作")


# 设计一个函数, 在函数内容创建对象
```

```
def foo():
    # 函数调用完毕，里面创建的对象会销毁，生命周期结束，自动调用__del__方法
    dog1 = Dog()


print("函数被调用前")
# 调用函数
foo()
print("函数被调用后")
```

- 对象销毁[扩展]:

```
"""
在对象的生命周期结束(对象销毁)时，__del__()方法会自动被调用，做一些清理工作
"""


class Dog(object):

    def __del__(self):
        print("对象被销毁了,做清理动作")


# 设计一个函数，在函数内容创建对象
```

```
def foo():
    # 函数调用完毕，里面创建的对象会销毁，生命周期结束，自动调用__del__方法
    dog1 = Dog()


print("函数被调用前")
# 1.调用函数： 函数调用结束,函数中创建对象会被销毁
foo()
print("函数被调用后")

print("创建对象前")
dog2 = Dog()
del dog2    # 2.del 对象， 销毁对象.
print("销毁对象后")

dog3 = Dog()    # 3.程序结束后,所有对象会被销毁
```

4.【应用】烤地瓜

4.1. 普通版本

```
"""
# SweetPotato 类的设计
    地瓜有两个属性:
        状态 state: 字符串
        烧烤总时间 cooked_time: 整数

# 1. 定义__init__方法, 添加2个属性
    # 1.1 默认状态state是生的
    # 1.2 默认时间cooked_time是0

# 2. 定义__str__方法
    # 2.1 返回地瓜状态, 烧烤总时间

# 3. 定义 cook 方法, 提供参数 time 设置 本次烧烤的时间
    # 3.1 使用 本次烧烤时间 对 烧烤总时间 进行累加
    # 3.2 根据 烧烤总时间, 设置地瓜的状态:
        [0, 3) -> 生的
        [3, 6) -> 半生不熟
        [6, 8) -> 熟了
        大于等于8 -> 烤糊了

# 4. 主逻辑程序
```

```
# 4.1 创建 地瓜对象
# 4.2 分多次烧烤地瓜
# 4.3 每烧烤一次，输出地瓜信息
"""


# SweetPotato 类的设计
#     地瓜有两个属性:
#         状态 state: 字符串
#         烧烤总时间 cooked_time: 整数

class SweetPotato(object):

    # 1. 定义__init__方法，添加2个属性
    def __init__(self):
        # 1.1 默认状态state是生的
        self.state = "生的"
        # 1.2 默认时间cooked_time是0
        self.cooked_time = 0

    # 2. 定义__str__方法
    def __str__(self):
        # 2.1 返回地瓜状态，烧烤总时间
        return f"地瓜状态: {self.state} 烧烤
总时间: {self.cooked_time}"
```

```
    # 3. 定义 cook 方法, 提供参数 time 设置 本次烧烤的时间
    def cook(self, time):
        # 3.1 使用 本次烧烤时间 对 烧烤总时间 进行 累加
        self.cooked_time += time
        # 3.2 根据 烧烤总时间, 设置地瓜的状态:
        if self.cooked_time >= 0 and self.cooked_time < 3:
            #     [0, 3) -> 生的
            self.state = "生的"
        elif self.cooked_time >= 3 and self.cooked_time < 6:
            #     [3, 6) -> 半生不熟
            self.state = "半生不熟"
        elif self.cooked_time >= 6 and self.cooked_time < 8:
            #     [6, 8) -> 熟了
            self.state = "熟了"
        elif self.cooked_time >= 8:
            #     大于等于8 -> 烤糊了
            self.state = "烤糊了"


# 4. 主逻辑程序
# 4.1 创建 地瓜对象
```

```
sp = SweetPotato()
print(sp)

# 4.2 分多次烧烤地瓜
sp.cook(2)
print(sp)    # # 4.3 每烧烤一次，输出地瓜信息

sp.cook(3)
print(sp)    # 4.3 每烧烤一次，输出地瓜信息

sp.cook(2)
print(sp)    # 4.3 每烧烤一次，输出地瓜信息

sp.cook(1)
print(sp)    # 4.3 每烧烤一次，输出地瓜信息
```

4.2. 拓展版本

- 烤地瓜案例的实现思路：

```
"""
# SweetPotato 类的设计
    地瓜有两个属性：
        状态 state：字符串
        烧烤总时间 cooked_time：整数
```

```
# 1. 定义__init__方法，添加2个属性
    # 1.1 默认状态state是生的
    # 1.2 默认时间cooked_time是0

# 2. 定义__str__方法
    # 2.1 返回地瓜状态，烧烤总时间

# 3. 定义 cook 方法，提供参数 time 设置 本次烧烤的时间
    # 3.1 使用 本次烧烤时间 对 烧烤总时间 进行累加
    # 3.2 根据 烧烤总时间，设置地瓜的状态：
        [0, 3) -> 生的
        [3, 6) -> 半生不熟
        [6, 8) -> 熟了
        大于等于8 -> 烤糊了

# 4. 主逻辑程序
# 4.1 创建 地瓜对象
# 4.2 分多次烧烤地瓜
# 4.3 每烧烤一次，输出地瓜信息

# 5. 拓展功能
# 5.1 添加属性 condiments， 列表类型，默认为空列表
```

```
# 5.2 修改 __str__ 返回信息，返回增加已添加的佐料信息
# 5.3 定义 add_condiments(self, temp), temp为添加什么佐料的参数
    # 5.3.1 佐料列表追加元素
# 5.4 再次测试代码，添加佐料，重新打印信息
"""
```

代码:

```
"""
# SweetPotato 类的设计
    地瓜有两个属性:
        状态 state: 字符串
        烧烤总时间 cooked_time: 整数

# 1. 定义__init__方法，添加2个属性
    # 1.1 默认状态state是生的
    # 1.2 默认时间cooked_time是0

# 2. 定义__str__方法
    # 2.1 返回地瓜状态，烧烤总时间

# 3. 定义 cook 方法，提供参数 time 设置 本次烧烤的时间
    # 3.1 使用 本次烧烤时间 对 烧烤总时间 进行累加
```

```
    # 3.2 根据 烧烤总时间，设置地瓜的状态：
        [0，3) -> 生的
        [3，6) -> 半生不熟
        [6，8) -> 熟了
        大于等于8 -> 烤糊了

# 4. 主逻辑程序
# 4.1 创建 地瓜对象
# 4.2 分多次烧烤地瓜
# 4.3 每烧烤一次，输出地瓜信息

# 5. 拓展功能
# 5.1 添加属性 condiments， 列表类型，默认为空列表
# 5.2 修改 __str__ 返回信息，返回增加已添加的佐料信息
# 5.3 定义 add_condiments(self，temp)，temp为添加什么佐料的参数
    # 5.3.1 佐料列表追加元素
# 5.4 再次测试代码，添加佐料，重新打印信息

"""

# SweetPotato 类的设计
#     地瓜有两个属性：
#         状态 state：字符串
```

```
#           烧烤总时间 cooked_time: 整数


class SweetPotato(object):

    # 1. 定义__init__方法，添加2个属性
    def __init__(self):
        # 1.1 默认状态state是生的
        self.state = "生的"
        # 1.2 默认时间cooked_time是0
        self.cooked_time = 0
        # 5.1 添加属性 condiments，  列表类型，默认为空列表
        self.condiments = []

    # 2. 定义__str__方法
    def __str__(self):
        # 2.1 返回地瓜状态，烧烤总时间
        # 5.2 修改 __str__ 返回信息，返回增加已添加的佐料信息
        return f"地瓜状态: {self.state}, 烧烤总时间: {self.cooked_time} 分钟, 已添加的佐料信息: {self.condiments}"

    # 3. 定义 cook 方法，提供参数 time 设置 本次烧烤的时间
```

```
    def cook(self, time):
        # 3.1 使用 本次烧烤时间 对 烧烤总时间 进行 累加
        self.cooked_time += time
        # 3.2 根据 烧烤总时间, 设置地瓜的状态:
        if 0 <= self.cooked_time < 3:
            #     [0, 3) -> 生的
            self.state = "生的"
        elif 3 <= self.cooked_time < 6:
            #     [3, 6) -> 半生不熟
            self.state = "半生不熟"
        elif 6 <= self.cooked_time < 8:
            #     [6, 8) -> 熟了
            self.state = "熟了"
        elif self.cooked_time >= 8:
            #     大于等于8 -> 烤糊了
            self.state = "烤糊了"

    # 5.3 定义 add_condiments(self, temp), temp为添加什么佐料的参数
    def add_condiments(self, temp):
        # 5.3.1 佐料列表追加元素
        self.condiments.append(temp)


# 5.4 再次测试代码, 添加佐料, 重新打印信息
```

```
sp = SweetPotato()
print(sp)

sp.cook(2)
sp.add_condiments("蜜糖")
print(sp)

sp.cook(3)
sp.add_condiments("孜然")
print(sp)

sp.cook(2)
sp.add_condiments("烧烤酱")
print(sp)
```

5.【应用】搬家具

5.1. 搬家具普通版本

```
"""
家具类 Item
# 1. 定义__init__方法，添加2个属性，需要2个形参 _type, _area
```

```
    # 1.1 家具类型 type
    # 1.2 家具面积 area

# 2. 实现__str__方法
    # 2.1 返回家具类型和家具面积

房子类 Home
# 1. 定义__init__方法，添加3个属性，需要3个形参
    # 1.1 地址 address
    # 1.2 房子面积 area
    # 1.3 房子剩余面积 free_area，默认为房子的面积

# 2. 实现__str__方法
    # 2.1 返回房子地址、面积、剩余面积信息

# 3. 实现add_item方法，提供item参数来添加家具，item是对象
    # 3.1 如果 房间的剩余面积 >= 家具的面积，可以容纳家具：
        # 3.1.1 打印添加家具的类型和面积
        # 3.1.2 剩余面积 减少
    # 3.2 否则 不能容纳家具：提示家具添加失败

主程序逻辑：
```

```
# 1. 创建 家具对象，输出 家具信息
# 2. 创建 房子对象，输出 房子信息
# 3. 房子添加家具，输出 房子信息
"""


# 家具类 Item
class Item(object):
    # 1. 定义__init__方法，添加2个属性，需要2个形参 _type, _area
    def __init__(self, _type, _area):
        # 1.1 家具类型 type
        self.type = _type
        # 1.2 家具面积 area
        self.area = _area

    # 2. 实现__str__方法
    def __str__(self):
        # 2.1 返回家具类型和家具面积
        return f"家具的类型：{self.type}，家具的面积：{self.area} 平米"


 # 房子类 Home
class Home(object):
```

```
    # 1. 定义__init__方法，添加3个属性，需要3个形参
    def __init__(self, _address, _area):
        # 1.1 地址 address
        self.address = _address
        # 1.2 房子面积 area
        self.area = _area
        # 1.3 房子剩余面积 free_area，默认为房子的面积
        self.free_area = _area

    # 2. 实现__str__方法
    def __str__(self):
        # 2.1 返回房子地址、面积、剩余面积信息
        return f"房子地址：{self.address}，面积：{self.area}平米，剩余面积：{self.free_area}平米"

    # 3. 实现add_item方法，提供temp参数来添加家具，temp是家具类型的对象
    def add_item(self, temp):
        # 3.1 如果 房间的剩余面积 >= 家具的面积，可以容纳家具：
        if self.free_area >= temp.area:
            # 3.1.1 打印添加家具的类型和面积
            print("家具添加成功，", temp)
```

```
                # 3.1.2 剩余面积 减少
                self.free_area -= temp.area
            # 3.2 否则 不能容纳家具：提示家具添加失败
            else:
                print("家具添加失败")


it1 = Item("大床", 4)
print(it1)

h1 = Home("北京一环四合院", 160)
print(h1)

h1.add_item(it1)
print(h1)
```

5.2. 搬家具拓展版本

- 搬家具案例的实现思路：

```
"""
家具类 Item
# 1. 定义__init__方法，添加2个属性，需要2个形参 _type, _area
    # 1.1 家具类型 type
```

```
    # 1.2 家具面积 area

# 2. 实现__str__方法
    # 2.1 返回家具类型和家具面积

房子类 Home
# 1. 定义__init__方法，添加3个属性，需要3个形参
    # 1.1 地址 address
    # 1.2 房子面积 area
    # 1.3 房子剩余面积 free_area，默认为房子的面积

# 2. 实现__str__方法
    # 2.1 返回房子地址、面积、剩余面积信息

# 3. 实现add_item方法，提供item参数来添加家具，item是对象
    # 3.1 如果 房间的剩余面积 >= 家具的面积，可以容纳家具：
        # 3.1.1 打印添加家具的类型和面积
        # 3.1.2 剩余面积 减少
    # 3.2 否则 不能容纳家具：提示家具添加失败

主程序逻辑：
# 1. 创建 家具对象，输出 家具信息
```

```
# 2. 创建 房子对象，输出 房子信息
# 3. 房子添加家具，输出 房子信息

输出房子时，显示包含的所有家具的类型
# a. Home类中添加 item_type_list 属性(家具类型列表)，用于记录所有家具类型
# b. Home类的 add_item 方法中，将添加成功的家具类型 添加到 item_type_list 中
# c. Home类的 __str__ 方法中，打印家具的类型列表
"""
```

- 代码

```
"""
家具类 Item
# 1. 定义__init__方法，添加2个属性，需要2个形参 _type, _area
    # 1.1 家具类型 type
    # 1.2 家具面积 area

# 2. 实现__str__方法
    # 2.1 返回家具类型和家具面积

房子类 Home
# 1. 定义__init__方法，添加3个属性，需要3个形参
```

```
    # 1.1 地址 address
    # 1.2 房子面积 area
    # 1.3 房子剩余面积 free_area，默认为房子的面积

# 2. 实现__str__方法
    # 2.1 返回房子地址、面积、剩余面积信息

# 3. 实现add_item方法，提供item参数来添加家具，item是对象
    # 3.1 如果 房间的剩余面积 >= 家具的面积，可以容纳家具：
        # 3.1.1 打印添加家具的类型和面积
        # 3.1.2 剩余面积 减少
    # 3.2 否则 不能容纳家具：提示家具添加失败

主程序逻辑：
# 1. 创建 家具对象，输出 家具信息
# 2. 创建 房子对象，输出 房子信息
# 3. 房子添加家具，输出 房子信息

输出房子时，显示包含的所有家具的类型
# a. Home类中添加 item_type_list 属性(家具类型列表)，用于记录所有家具类型
# b. Home类的 add_item 方法中，将添加成功的家具类型 添加到 item_type_list 中
```

```
# c. Home类的 __str__ 方法中，打印家具的类型列表
"""


# 家具类 Item
class Item(object):
    # 1. 定义__init__方法，添加2个属性，需要2个形参 _type, _area
    def __init__(self, _type, _area):
        # 1.1 家具类型 type
        self.type = _type
        # 1.2 家具面积 area
        self.area = _area

    # 2. 实现__str__方法
    def __str__(self):
        # 2.1 返回家具类型和家具面积
        return f"家具的类型: {self.type}, 家具的面积: {self.area} 平米"


# 房子类 Home
class Home(object):
    # 1. 定义__init__方法，添加3个属性，需要3个形参
```

```
    def __init__(self, _address, _area):
        # 1.1 地址 address
        self.address = _address
        # 1.2 房子面积 area
        self.area = _area
        # 1.3 房子剩余面积 free_area，默认为房子的面积
        self.free_area = _area
        # a. Home类中添加 item_type_list 属性(家具类型列表)，用于记录所有家具类型
        self.item_type_list = []

    # 2. 实现__str__方法
    def __str__(self):
        # 2.1 返回房子地址、面积、剩余面积信息
        # c. Home类的 __str__ 方法中，打印家具的类型列表
        return f"房子地址：{self.address}，面积：{self.area}平米，剩余面积：{self.free_area}平米，房子中添加的家具有：{self.item_type_list}"

    # 3. 实现add_item方法，提供temp参数来添加家具，temp是家具类型的对象
    def add_item(self, temp):
```

```
        # 3.1 如果 房间的剩余面积 >= 家具的面积，可以容纳家具：
        if self.free_area >= temp.area:
            # 3.1.1 打印添加家具的类型和面积
            print("家具添加成功, ", temp)
            # b. Home类的 add_item 方法中，将添加成功的 家具类型 添加到 item_type_list 中

 self.item_type_list.append(temp.type)
            # 3.1.2 剩余面积 减少
            self.free_area -= temp.area
        # 3.2 否则 不能容纳家具：提示家具添加失败
        else:
            print("家具添加失败")


h1 = Home("北京一环四合院", 160)
print(h1)

tv = Item("大电视", 3)
print(tv)

h1.add_item(tv)
print(h1)
```

```
it1 = Item("游泳池", 200)
h1.add_item(it1)
print(h1)
```