Python魔法方法：

一、什么是魔法方法？
1、魔法方法名前后都有两个下划线 __init__ 、__new__ 、__del__ 、__str__
2、魔法方法无需手动调用，会自己后台执行。

二、魔法方法学习目标？
认识什么是魔法方法，并且认识几个常见的魔法方法。

三、常见魔法方法：
__init__ 初始化方法 ： 进行对象初始化
__new__ 构造方法 ： 创建一个对象。 __new__ 是默认在创建对象的时候自动执行的。一般情况下我们不需要去对其进行操作。
del(self) 析构方法，是一个特殊方法，用于在Python对象被删除时执行一些清理工作。当对象不再被使用时，Python的垃圾回收机制会自动调用__del__方法。方法
__str__(self)：用于返回对象的字符串内容。它被称为"对象的可打印表示"，因为它通常用于打印对象或将对象转换为字符串。

在 Python 中，创建一个新对象的基本过程是：
1、调用__new__方法创建新对象
2、返回对象给 __init__方法
3、在 __init__ 方法中初始化对象的属性

```
# 定义类的时候，类名后面的括号可以写可不写，只要我们没有给该类指明父类，它会自动继承Object类。
Object类 是所有类的父类
class MyClass():
   # 方法重写，因为我们继承了Object类，所以也就继承了Object类中的new方法
    def __new__(cls, *args, **kwargs):
        print("正在创建对象")
        return super().__new__(cls  #创建对象，并返回给__init__方法
    def __init__(self,classId,className):     # 初始化方法
        print("正在进行初始化")
        self.classId = classId       # 设置属性值
        self.className = className     # 初始化值
        print(f"初始化完成！classId= {classId},className={className}")
    # 对象的可打印表示
    def __str__(self):
        return f'当前类属性classId= {self.classId},className={self.className}'
    # 析构方法
    def __del__(self):
        print("马上结束！")
"    程序在运行过程中会去占用电脑内存和各种资源，我们Python代码执行也是一样会消耗资源。 python提供了一个垃圾回收机制，这个机制可以帮助我们自动在程序结束后清理数据，回收电脑资源。"
# 创建了一个对象，并给改对象设置初始化值（初始化）
x = MyClass('001','vip19')
print(x)
```

装饰器

装饰器其实也是一个闭包，其功能就是在不改动目标函数的同时，增加额外的功能。

```
'''复习：
什么是闭包？
当两个函数进行嵌套的时候，外层函数返回内层函数的引用，这种现象叫做闭包'''
def x():
    def func():
        print("我是一个闭包",f"{a},{b},{c}")
    return func           # y这个名字代表 y函数 他是一个引用， y()则是对y函数的调用
x()()
```

写装饰器的步骤

```
"""
总结：
0、目标升级函数
1、写闭包->进行正常调用验证能否实现目标效果
2、给原函数加 @ 装饰器 ，让闭包去装饰 原函数，增加其功能。之后我们再去调用原函数的时候，执行结果会进行升级。
"""
```

1、确定原函数 【实现一个简单的睡眠功能】 睡觉1.0

```
def sleep():
    print("睡觉")
```

新需求： *升级 2.0版本，在睡眠之前进行提示即将开始睡醒，睡眠之后，提示睡够8个小时马上起床。不能改基础源码。*

2、写闭包（在写的时候可以调试代码、测试功能）【实现2.0功能】

```
def add_sleep(func):      # func是形参名字
    def add():      #写实际2.0版本业务的函数
        print("即将开始睡眠")    # 新功能
        func()     #调用旧功能
        print("睡够8小时，即将起床！")      #新功能
    return add     #返回2.0业务功能的引用

# 测试
sleep2_0 = add_sleep(sleep)
sleep2_0()
# add_sleep(sleep)()  # 闭包调用简写
" 目前来说还没有真正意义上实现不改变原代码增加其功能 。 1.0已经被别的地方使用了，所以我希望能够不改变别的地方的使用代码，也就是说，我这个1.0的名字具有2.0的功能"
```

3、将装饰器语法糖加上 【@会将1.0当做参数传递给2.0闭包，然后返回2.0的功能引用】

```
@add_sleep       #装饰器语法糖
def sleep():
    print("睡觉")
```

完整代码：

```
# add_sleep 是 sleep的装饰器
def add_sleep(func):      # func是形参名字
```

```
    def add():      #写实际2.0版本业务的函数
        print("即将开始睡眠")    # 新功能
        func()     #调用旧功能
        print("睡够8小时，即将起床！")      #新功能
    return add     #返回2.0业务功能的引用

@add_sleep       #装饰器语法糖
def sleep():
    print("睡觉")

@add_sleep
def heShui():
    print("喝水")

sleep()
heShui()
```

设计模式：

设计模式是一种在软件设计中经过验证的解决方案，它提供了在特定情境下解决常见问题的指导原则和实践。设计模式是由软件开发领域的专家们根据他们的经验和最佳实践总结出来的。面向对象就是比较经典的设计模式。

除了面向对象外，在编程中也有很多既定的套路可以方便开发，我们称之为设计模式:

- 单例、工厂模式
- 建造者、责任链、状态、备忘录、解释器、访问者、观察者、中介、模板、代理模式
- 等等模式

单例模式：

类创建实例后，就可以得到一个完整的，独立的类对象。通过print语句我们可以看到每个对象的内存地址是不相同的。

某些场景下，我们需要一个类无论获取多少次类对象，都仅仅只提供一个具体的实例用以节省创建类对象的开销和内存开销。

比如某些工具类，仅需要1个实例，即可在各处使用。这就是单例模要实现的效果。

```
使用场景:当一个类只能有一个实例，而客户可以从一个众所周知的访问点访问它时。
优点：
    1、节省内存                          2、节省创建对象的开销
```

实现方式一：

定义:保证一个类只有一个实例,并提供一个访问它的全局访问点

```
#定义工具类
class 工具类():
    pass
#创建公共工具实例
gongJu = 工具类()
#设定访问它的全局访问点
def Gong_ju():
    return gongJu
#创建对象实例
```

```
g1 = Gong_ju()
g2 = Gong_ju()
#查看创建实例的内存地址
print(id(g1))
print(id(g1))
```

实现方式二：

```
class 工具类():  # 在开发中只需要一个工具对象
    __gongju = None   # 私有变量，用于存放已经创建的实例
    def __new__(cls, *args, **kwargs):
        if cls.__gongju:     # 判断 __cgongju里面是否已经有实例对象
            return cls.__gongju
        else:
            cls.__gongju = super().__new__(cls)    #  创建一个对象实例
            return cls.__gongju
    def __init__(self):
        print(self.__gongju)
```

工厂模式：

定义：将对象的创建由使用原生类本身创建。

当需要大量创建一个类的实例的时候，可以使用工厂模式。即，从原生的使用类的构造去创建对象的形式迁移到，基于工厂提供的方法去创建对象的形式。

优点：

1、大批量创建对象的时候有统一的入口，易于代码维护

2、当发生修改，仅修改工厂类的创建方法即可

3、符合现实世界的模式，即由工厂来制作产品(对象)

```python
class Person:
    pass

class Worker(Person):
    pass
class Student(Person):
    pass
class Teacher(Person):
    pass

class Factory:
    def get_person(self, p_type):
        if p_type == 'w':
            return Worker()
        elif p_type == 's':
            return Student()
        else:
            return Teacher()

factory = Factory()
worker = factory.get_person('w')
stu = factory.get_person('s')
teacher = factory.get_person('t')
```