面向对象编程

编程范式

> 编程是程序猿使用数据结构+算法，通过特定的编程语言组成的代码来告诉计算机如何执行任务。条条大路通罗马，每个程序员解决同样的问题代码几乎都不相同，每一种编程任务会有很多条实现方法。对这些不同的编程方式的特点进行归纳总结得出来的编程方式类别，即为编程范式。不同的编程范式本质上代表对各种类型的任务采取的不同的解决问题的思路，大多数语言只支持一种编程范式，当然也有些语言可以同时支持多种编程范式。两种最重要的编程范式分别是面向过程编程和面向对象编程。

面向过程编程(Procedural Programming)

> Procedural programming uses a list of instructions to tell the computer what to do step-by-step.
> 面向过程编程就是程序从上到下一步步执行，从上到下，从头到尾的解决问题 。基本设计思路就是程序一开始是要着手解决一个大的问题，然后把一个大问题分解成很多个小问题或子过程，这些子过程再执行的过程再继续分解直到小问题足够简单到可以在一个小步骤范围内解决。

面向对象编程（OOP——Object Oriented Programming)

> OOP编程是利用"类"和"对象"来创建各种模型来实现对真实世界的描述，使用面向对象编程的原因一方面是因为它可以使程序的维护和扩展变得更简单，并且可以大大提高程序开发效率 ，另外，基于面向对象的程序可以使它人更加容易理解你的代码逻辑，从而使团队开发变得更加方便有效率。

类的基础语法

```
class Policeman:

    def __init__(self):
        print("Start!>>>>>")
        self.name = "Jack"
        self.gender = "male"
        self.skill = "翻跟斗"

p1 = Policeman()

控制台输出结果:
Start!>>>>>
```

注:

- 类名要大写
- `__init__`函数名称是固定的
- `__init__`函数必须传递一个参数`self`
- `__init__`函数是在对象创建时自动执行的

属性的修改和删除

```python
p1 = Policeman()
print(p1.__dict__)
print(p1.__dict__["name"])
print(p1.name)

p1.gender = "不详"
print(p1.gender)

del p1.gender
print(p1.__dict__)
```

既然`__int__`作为一个函数，类是一个模板，那么，可以在创建对象时传递类的参数

```python
class Policeman:

    def __init__(self, name, gender, skill):
        print("开始>>>>>>")
        self.name = name
        self.gender = gender
        self.skill = skill
        print(self.__dict__)

    def job(self):
        print("%s抓小偷" % self.name)

Jack = Policeman("Jack", "male", "翻跟斗")
print(Jack.name, Jack.gender, Jack.skill)

Alice = Policeman("Alice", "femal", "跳舞")
Alice.job()
```

如果类中的函数名称不是`__init__`，那么需要调用才可以执行，而不会自动执行

Python中创建一个类时，自动开辟一块内存空间，我们之前给类设置属性时，使用`__init__`方法

```python
class Policeman:

    country = "China"

    def __init__(self, name, gender, skill):

        self.name = name
        self.gender = gender
        self.skill = skill

Jack = Policeman("Jack", "male", "翻跟斗")
print(Policeman.country)
```

- country变量是一个`静态属性`，存储在Policeman类中
- 当创建一个Jack对象时，将Policeman类的指针保存在Jack中，这样对象和类才可以关联起来

```python
class Policeman:
```

```
    country = "China"

    def __init__(self, name, gender, skill, country):

        self.name = name
        self.gender = gender
        self.skill = skill
        self.country = country

Jack = Policeman("Jack", "male", "翻跟斗", "USA")
print(Policeman.country)
print(Jack.country)
```

- 修改类中的静态属性时，必须使用类名而不能使用对象名
- 在init方法中，传递了country参数，使用Jack对象调用country属性时，首先调用自己内存空间中的country，如果init方法中没有传递country参数，就会到类的空间中寻找country

什么时候使用静态变量?

```
    如果一个变量是所有对象共享的值，那么这个变量就应该被定义为静态变量
```

练一练：计数，一个类创建了多少个对象，创建一个对象，计数加1

```
class Policeman:

    country = "China"
    count = 0

    def __init__(self, name, gender, skill, country):

        self.name = name
        self.gender = gender
        self.skill = skill
        self.country = country
        Policeman.count += 1

Jack = Policeman("Jack", "male", "翻跟斗", "USA")
print(Policeman.country)
print(Jack.country)
print(Policeman.count)

Alice = Policeman("Alice", "femal", "跳舞", "Russia")
print(Policeman.count)
```

面向对象继承

继承（inheritance）：是面向对象软件技术中的一个概念。如果一个类A继承自另一个类B，就把这个A称为B的子类别，把B称为A的父类别或者超类。继承可以使子类具有父类的各种属性和方法，而不再需要编写相同的代码。在令子类继承父类的同时，可以重新定义某些属性，并重新某些方法，即覆盖父类别原有属性和方法，使其获得与父类别不同的功能，可以很好地提高代码的复用性、扩展性。

```python
class Person:

    def __init__(self, name, gender, skill):

        self.name = name
        self.gender = gender
        self.skill = skill

class Policeman(Person):

    def job(self):

        print("%s的工作是抓小偷" % self.name)

class Thief(Person):

    def job(self):

        print("%s的工作是偷东西" % self.name)

Jack = Policeman("Jack", "male", "翻跟斗")
Jack.job()

Sam = Policeman("Sam", "male", "翻跟斗")
Sam.job()
```

多继承

```python
class YangJian:

    def weopon(self):
        print('三尖两刃刀 + 哮天犬')

class WuKong:

    def skill(self):
        print('七十二变 + 筋斗云')

class BlackCat(YangJian, WuKong):

    pass


cat = BlackCat()
cat.weopon()
cat.skill()
```

子类调用父类方法

```python
class Person:

    def __init__(self, name, gender, skill):
```

```
        self.name = name
        self.gender = gender
        self.skill = skill

class Policeman(Person):

    def __init__(self, name, gender, skill, country):
        super().__init__(name, gender, skill)
        self.country = country

    def output(self):
        print(self.name, self.gender, self.skill, self.country)


class Thief(Person):

    def __init__(self, name, gender, skill, country):
        # super().__init__(name, gender, skill)
        Person.__init__(self, name, gender, skill)
        self.country = country

    def output(self):
        print(self.name, self.gender, self.skill, self.country)

Jack = Policeman('Jack', 'm', '翻跟斗', '巨人国')
Jack.output()

Sam = Thief('Sam', 'm', '翻墙', '小人国')
Sam.output()
```

多态

> 同一个对象的多种形态

```
class Bird:

    def fly(self):
        print("小鸟在天空飞翔")

class Plane:

    def fly(self):
        print("飞机在天空飞翔")

class Rocket:

    def fly(self):
        print("火箭飞向太空")

def fly(obj):

    obj.fly()

bird = Bird()
```

```
plane = Plane()
rocket = Rocket()
```

注：Python中虽然支持多态，但是有限的支持多态，也不支持运算符重载

封装

> 封装是指将功能模块化，比如，我们写一个求和函数就是封装，函数使用者不需要了解函数内部是如何实现求和的，只需要调用我们写好的函数就可以了。把很多数据封装到一个对象中，把固定功能的代码封装到一个代码块，将函数、对象打包成模块，这些都属于封装思想。

```
class Person:

    def __init__(self, name, gender, skill):

        self.__name = name
        self.__gender = gender
        self.__skill = skill

    def output(self):
        print(self.__name, self.__gender, self.__skill)

p1 = Person('小明', 'm', 10)
# print(p1.__name)
p1.output()
print(p1._Person__name, p1._Person__gender, p1._Person__skill)
```

> 注：在子类中定义的__XXX不会覆盖父类中定义的__XXX，因为子类中变形成了：_子类名__XXX，而父类中变形成了_父类名__XXX。

面向对象进阶

类的约束

```
class Payment:

    def pay(self, money):
        raise Exception("你必须重写pay方法")


class QQpay(Payment):

    def pay(self, money):
        print("QQ支付%d元" % money)

class Wechatpay(Payment):

    def pay(self, money):
        print("微信支付%d元" % money)
```

```
16
17 class Alipay(Payment):
18
19     pass
20
21 qq = QQpay()
22 wx = Wechatpay()
23 zfb = Alipay()
24
25 qq.pay(10)
26 wx.pay(20)
27 zfb.pay(30)
```

```
1  from abc import ABCMeta, abstractmethod
2
3  class Payment(metaclass=ABCMeta):
4
5      @abstractmethod
6      def pay(self, money):
7          print("继承我必须得重写")
8
9
10 class QQpay(Payment):
11
12     def pay(self, money):
13         print("QQ支付%d元" % money)
14
15 class Wechatpay(Payment):
16
17     def pay(self, money):
18         print("微信支付%d元" % money)
19
20 class Alipay(Payment):
21
22     pass
23
24 qq = QQpay()
25 wx = Wechatpay()
26 # zfb = Alipay()
27
28 def pay(obj, money):
29     obj.pay(money)
30
31 pay(qq, 10)
32 pay(wx, 20)
33 # pay(zfb, 30)
```

> 使用抛出异常方法，更加明确，也更加专业

类方法classmethod

类方法通过**@classmethod装饰器**实现，类方法和普通方法的区别是， 类方法**只能访问类变量**，不能访问实例变量

```python
class Animal:

    __feature = "delicious"
    country = "China"

    def __init__(self, name, color):

        self.name = name
        self.color = color

    @classmethod
    def get_feature(cls):

        print("所有的动物都很%s，尤其是bat" % cls.name)

Animal.get_feature()
```

静态方法staticmethod

静态方法是类中的函数，通过@staticmethod装饰器实现，不需要实例。静态方法主要是用来存放逻辑性的代码，逻辑上属于类，但是和类本身没有关系，也就是说在静态方法中，不会涉及到类中的属性和方法的操作。可以理解为，静态方法是个**独立的、单纯的**函数，它仅仅托管于某个类的名称空间中，便于使用和维护。

```python
import time

class Time:

    def __init__(self, hour, minute, second):

        self.hour = hour
        self.minute = minute
        self.second = second

    @staticmethod
    def current_time():

        print("当前时间戳为：%s" % time.time())


Time.current_time()

# t1 = Time(5, 30, 20)
# Time.current_time(t1)
```

面向对象property属性

遵循了统一访问的原则

```python
import math
```

```python

class Circle:

    def __init__(self, r):

        self.__r = r

    @property
    def area(self):

        return round(self.__r**2 * math.pi, 2)

    @area.setter
    def area(self, r):

        self.__r = r

    @area.deleter
    def area(self):

        del self.__r
        print("我删除了")

c1 = Circle(5)
print("c1的面积是: ", c1.area)

c1.area = 6
print("c1的面积是: ", c1.area)

del c1.area

print("c1的面积是: ", c1.area)
```

反射

反射的概念是由Smith在1982年首次提出的，主要是指程序可以访问、检测和修改它本身状态或行为的一种能力（自省）。这一概念的提出很快引发了计算机科学领域关于应用反射性的研究。它首先被程序语言的设计领域所采用,并在Lisp和面向对象方面取得了成绩。

Python面向对象中的反射：通过字符串的形式操作对象相关的属性。

```python
class Fruit:

    def __init__(self, name, color):

        self.name = name
        self.color = color

    def buy(self, price, num):

        print(price * num)

apple = Fruit("苹果", "红色")

print(hasattr(apple, "name"))
```

```
print(hasattr(apple, "buy"))

print(getattr(apple, "name"))

f = getattr(apple, "buy")
f(5, 10)

delattr(apple, "name")
print(hasattr(apple, "name"))
```

```
class WebSite:

    def register(self):
        print("欢迎注册")

    def login(self):
        print("欢迎登陆")

    def home(self):
        print("欢迎进入首页")

    def about(self):
        print("关于我们")

while True:

    choose = input("请输入>>>")

    if choose == "register":
        page = WebSite()
        page.register()

    elif choose == "login":
        page = WebSite()
        page.login()

    elif choose == "home":
        page = WebSite()
        page.home()

    elif choose == "about":
        page = WebSite()
        page.about()
```

```
class WebSite:

    def register(self):
        print("欢迎注册")

    def login(self):
        print("欢迎登陆")

```

```
    def home(self):
        print("欢迎进入首页")

    def about(self):
        print("关于我们")

page = WebSite()

while True:

    choose = input("请输入>>>")
    if hasattr(page, choose):
        f = getattr(page, choose)
        f()
    else:
        print("输入页面没有找到：404")
```

单例设计模式__new__

设计模式

- 设计模式是前人工作的总结和提炼，针对某一特定问题的成熟解决方案
- 提高代码复用性、增强代码可靠性

单例设计模式

- 让类创建的对象，在内存中只有唯一的一个实例
- 每一次实例化生成的对象，内存地址是相同的

> 例如，一个系统中可以存在多个打印任务，但是只能有一个正在工作的任务；一个音乐播放器里可以播放很多音乐，但是一次只能播放一个音乐

__new__方法

- `__new__`方法是由object基类提供的内置静态方法
- 在内存中为对象分配空间
- 返回对象引用
- Python解释器获得对象引用后，将引用作为第一个出参数传递给`__init__`方法

```python
class Player:

    def __new__(self, *args, **kwgrgs):
        print("new执行了")

    def __init__(self):
        print("init执行了")

video1 = Player()
print(video1)

video2 = Player()
print(video2)
```

```python
class Player:

    __instance = None

    def __new__(cls, *args, **kwgrgs):
        print("new执行了")
        if cls.__instance is None:
            cls.__instance = super().__new__(cls)
        return cls.__instance

    def __init__(self):
        print("init执行了")

video1 = Player()
print(video1)

video2 = Player()
print(video2)
```

只执行一次`__init__`方法

```python
class Player:

    __instance = None
    __flag = False

    def __new__(cls, *args, **kwgrgs):
        print("new执行了")
        if cls.__instance is None:
            cls.__instance = super().__new__(cls)
        return cls.__instance

    def __init__(self):

        if not Player.__flag:

            print("init执行了")
            Player.__flag = True

```

```
video1 = Player()
print(video1)

video2 = Player()
print(video2)
```

__str__和__repr__

调用__str__情况

- print(obj)
- str(obj)
- 用%s占位

调用__repr__情况

- 如果没有找到__str__，就会调用__repr__
- 用%r占位
- repr(obj)

```
class Animal:

    def __init__(self, name, color):

        self.name = name
        self.color = color

    def __str__(self):

        ret = self.name + "是" + self.color
        return ret


bat = Animal("蝙蝠", "黑色")
print(bat)
print(str(bat) + "口感很好")
print("蝙蝠的特征：%s" % bat)
```

```
class Animal:

    def __init__(self, name, color):

        self.name = name
        self.color = color

    def __repr__(self):

```

```
        ret = self.name + "是" + self.color
        return ret

bat = Animal("蝙蝠", "黑色")
print(bat)
print(str(bat) + "口感很好")
print("蝙蝠的特征：%s" % bat)

print(repr(bat))
print("你了解蝙蝠吗？ %r" % bat)
```