面向对象编程

- 讲师:潘sir

类和实例

两种基本程序设计思想：面向过程编程，面向对象编程

面向过程

- 面向过程：以事件为中心，编程的时候，把解决问题的步骤分析出来，然后用函数去实现这些步骤，逐步调用

```
# 面向对象的编程
# 1. 定义数据/变量
std1={"name": "duoceshi", "score": 100}
std2={"name": "阿强", "score": 90}
# 2. 定义函数
def print_score(std):
    print("step1")
    print("step2")
    print(std["name"], std["score"])
# 3. 调用函数
print_score(std1)
print_score(std2)
```

```
step1
step2
duoceshi 100
step1
step2
阿强 90
```

面向对象

- 面向对象：首要考虑的是对象(类)，比如：把学生抽象成一个对象，它name和score属性，它也有print_score方法，使用对象来解决问题的思路：
    1. 先定义一个学生类
    2. 实例化：先把学生这个类给具体化，用name和score去创建这个对象
    3. 让这个对象，去执行 print_score方法，即可输出结果

类和实例

概念理解：

1. 类(class)：是一个抽象模版，比如：Student
2. 属性：类有很多特征，比如：Student有名字 和 成绩，对应的self.name就是类的属性，`__init__`就用来绑定属性的
3. 方法：类可以发出动作，比如：说出自己的成绩，对应的 print_score就是一个方法，类可以有很多方法
4. 对象/实例(instance)：把类具体化后，比如：a = 阿强/100分，就得到了一个具体对象，可以有多个对象，每个对象有不同的属性，但是有相同的方法
5. 实例化：把一个类，具体转换为一个对象，就叫做：实例化

```
# 通过class定义一个student类
# object可写，可不写，object是一个原始类，所有类都会继承它
# 名称一般大写开头，不是强制，这样好区分函数名
class Student(object):
    # 数据初始化，self是固定，后面的接收参数和函数的参数定义一样
    # 这行操作相当于把 name/score属性 绑定在了 Student类上
    def __init__(self, name, score=4):
        # 把属性绑定在类上
        self.x = "bangding"
        # 数据封装
        self.name = name
        self.score = score
    # 在类中定义方法，注意必须包含self，因为解释器会把自己传给print_score，如果不定义self参数，则会报错
    def print_score(self):
        # 直接应用类的属性变量，而不需要传参
        print('%s: %s' % (self.name, self.score))
        return 1
# 实例化，把类(模版) -> 具体化成一个对象(instance/实例)
dcs = Student("duocheshi")
aqiang = Student("阿强")
# dcs就是这个Student类的 一个实例，我们通过 duoceshi/100 对这个实例进行了初始化
print(dcs)
print(dcs.print_score())
```

```
<__main__.Student object at 0x106647748>
duocheshi: 4
1
```

`__init__` 解释说明：

1. `__init__` 方法的第一个参数永远是self，表示创建类实例本身
2. `__init__` 内部可以通过，self.x = xxx，把属性绑定在 类 上；self.name 就叫做**属性变量**(实例变量)
3. 属性变量可以接收外部变量，self.name = name 就是把外部变量传给属性变量
4. 一旦`__init__`方法定义了变量，那么创建实例的时候，就必须传合法的参数
5. 但是，self不需要传，python解释器会把自己当作一个对象，传进去

`def print_score(self):`解释说明：

1. 第一个参数必须是 实例变量 `self`
2. 调用参数的时候，不用传这个变量
3. 除了上面2点，其他和普通函数一样

面向过程/对象

- 面向过程：吃(狗, 屎)
- 面向对象：狗.吃(屎)

对比	面向过程	面向对象
优点	流程化，分步骤实现，效率高，代码短小精悍	结构化，模块化，容易扩展，可继承，可覆盖，低耦合，易维护
缺点	思考难度大，代码复用率低，扩展差，维护难	程序臃肿，开销大，性能低

小结

1. 类相当于模版，实例相当于一个个具体的对象，每个实例之间的数据互相独立
2. 方法指的是：与实例绑定的函数，方法可以通过属性变量直接访问实例的数据
3. 函数是直接调用，但方法必须在实例上进行调用
4. 调用方法可以直接操作对象内部数据，无需知道内部的实现细节，如：list.append() ''.join([]) xx.upper() tt.lower()

私有变量

为了防止调用者随意修改类变量，引入私有变量，注意：目的不是为了隐藏数据，而是为了对数据属性操作的严格控制。

```
class Student(object):

    def __init__(self, name, score):
        # 两个下划线，标识该变量为私有变量，只有内部可以访问，外部不能访问
        self.__name = name
        self.__score = score

    def print_score(self):
        print('%s: %s' % (self.__name, self.__score))

a = Student("阿强", 100)
# 不能访问私有变量，提示：没有该属性
a.__name
```

- 为了让外部只能访问，不能修改，可以新增方法来获取变量内容

```
class Student(object):

    def __init__(self, name, score):
        # 两个下划线，标识该变量为私有变量，只有内部可以访问，外部不能访问
        self.__name = name
        self.__score = score

    def get_name(self):
        return self.__name

    def get_score(self):
        return self.__score

    def print_score(self):
        print('%s: %s' % (self.__name, self.__score))

a = Student("阿强", 100)
# 通过方法来访问内部变量
a.get_name()
```

```
'阿强'
```

- 假如又需要更改内部变量，可以再引入一个set方法

```
class Student(object):

    def __init__(self, name, score):
        # 两个下划线，标识该变量为私有变量，只有内部可以访问，外部不能访问
        self.__name = name
        self.__score = score

    def get_name(self):
        return self.__name

    def get_score(self):
        return self.__score

    def set_name(self, x):
        self.__name = x

    def set_score(self, y):
        if 0 <= int(y) <= 100:
            self.__score = y
        else:
```

```
            raise ValueError("invalid score para")

    def print_score(self):
        print('%s: %s' % (self.__name, self.__score))

a = Student("阿强", 100)
# 通过方法来访问内部变量
a.get_name()
# 重设内部变量
a.set_name("阿珍")
a.get_name()
# 重写set方法，而不使用直接赋值，可以增加参数校验，保证数据准确
a.set_score(101)
```

注意：`__name__` 前后都有下划线的变量，是特殊变量，不是私有变量

课堂练习：把 age属性隐藏起来，用一个get方法，和一个set方法来代替

```
class Student(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

类的继承

- 子类继承父类

```
# 定义一个类
class Tester(object):
    def run(self):
        print("Tester newbee")

class AutoTester(Tester):
    pass
# 实例化
a = AutoTester()
# 子类继承父类的所有功能，可以直接调用
a.run()
```

```
Tester newbee
```

- 子类可以重写父类的方法

```
# 定义一个类
class Tester(object):
    def run(self):
        print("Tester newbee")

class AutoTester(Tester):
    # 重写方法
    def run(self):
        print("AutoTester newbee")
# 实例化
a = AutoTester()
a.run()
```

```
AutoTester newbee
```

- 定义class的时候，本质上就是定义了一种数据类型；数据类型的概念变大了

```
a = list() # a是list类型
b = Tester() # b是Tester类型
c = AutoTester() # c是AutoTester类型

# 判断对象的类型
isinstance(c, AutoTester)
# 子类也属于父类类型
# isinstance(c, Tester)
# 但父类不是子类的类型
isinstance(b, AutoTester)
```

```
False
```

多态

- 多态可以增加代码的灵活度，以继承和重写父类方法为前提，是一种调用方法的技巧，不会影响到类的内部设计

非多态模型

```
# 非多态模型
# 定义一个军犬类
class ArmyDog(object):

    def bite_enemy(self):
```

```
        print('追击敌人')
# 定义一个追毒犬类
class DrugDog(object):

    def track_drug(self):
        print('追查毒品')

# 定义人类
class Person(object):

    def work_with_army(self, dog):
        dog.bite_enemy()

    def work_with_drug(self, dog):
        dog.track_drug()

#
p = Person()
# 人带狗去咬人
p.work_with_army(ArmyDog())
# 人带狗去追毒
p.work_with_drug(DrugDog())
```

```
追击敌人
追查毒品
```

多态模型

从Dog派生出：AmyDog 和 DrugDog

```
# 定义一个父类
class Dog(object):
    def work(self):
        pass

# 定义子类，并重写方法
class ArmyDog(Dog):
    def work(self):
        print('追击敌人')

# 定义子类，并重写方法
class DrugDog(Dog):
    def work(self):
        print('追查毒品')

# 定义人类
class Person(object):
```

```
    def work_with_dog(self, dog):  # 只要能接收父类对象，就能接收子类对象
        dog.work()  # 只要父类对象能工作，子类对象就能工作。并且不同子类会产生不同的执行效果。


p = Person()
# 发现，只要调用父类的方法，传入不同的对象，可以获得不同的结果
p.work_with_dog(ArmyDog())
p.work_with_dog(DrugDog())
```

```
追击敌人
追查毒品
```

解释：

1. person在实现自己的方法时，直接调用dog父类的方法，如果传入dog子类，就会得到对应的子类结果
2. 有了多态，我们就不需要写针对不同的类写不同的方法，只需要针对父类写一个方法即可

```
# 方法会根据对象类型的不同，而获得不同的结果，而我们不需要写很多不同的 + 法
1 + 1
'a' + 'b'

xx = 'hello'
xx = [1, 2, 3]
for i in xx:
    print(i)
```

```
1
2
3
```

- 多态的优点：
    1. 增加了程序的灵活性：以不变应万变，不论对象千变万化，使用者都是同一种形式去调用，如dog.work()
    2. 增加了程序额可扩展性：通过继承dog类创建了一个新的类，使用者无需更改自己的代码，还是用dog.work()去调用

鸭子类型

如果走起路来像鸭子，叫起来也像鸭子，那么它就是鸭子

如果像编写一个现有对象的自定义版本，有两种方法：

1. 继承该对象，重写方法
2. 创建一个外观(属性)，和行为(方法)，都很像，但是和它无任何关系的全新对象

例子1: 模拟写出一个文件类，让json.load()可以操作它

```
import json
f = open('test_json.json', 'r')
print(f)
# 之所以能传入f对象，是因为f对象有read()  方法
print(json.load(f))



# 没有继承文件类，但看起来都像文件，因而就可以当文件一样去用
class TxtFile:
    def read(self):
        return r'{"hello": "duoceshi"}'
    def write(self):
        pass

a = TxtFile()
print(a)
print(json.load(a))
```

```
<_io.TextIOWrapper name='test_json.json' mode='r' encoding='UTF-8'>
{'hello': 'kitty'}
<__main__.TxtFile object at 0x10b600048>
{'hello': 'duoceshi'}
```

例子2：手动写一个迭代器

```
# 例子2：任何实现了 __iter__方法 和__next__方法的都是迭代器，还是可迭代对象
class iterator_cls(object):
    def __init__(self, num):
        self.num = num

    def __iter__(self):
        return self

    def __next__(self):
        self.num += 1
        if self.num > 15:
            raise StopIteration
        return self.num

from collections import Iterable
from collections import Iterator

if __name__ == '__main__':
    a = iterator_cls(10)
    print(isinstance(a, Iterable)) # True
```

```
    print(isinstance(a, Iterator)) # True
    # 可以迭代
    for i in a:
        print(i)
```

```
True
True
11
12
13
14
15
```

解释：

1. 前后都带__，在python里叫做：内置方法，也叫做魔法方法

总结

面向对象三大特性

1. 封装：根据职责，将属性和方法封装到一个抽象类中——定义类的准则
2. 继承：实现代码的重用，相同的代码不需要重复编写——设计类的技巧，子类针对自己的需求，编写特定的代码
3. 多态：不同的子类对象，调用相同的父类方法，产生不同的执行结果

类中的三种方法

1. 实例方法：第一个参数必须要默认传递实例对象，一般使用self。
2. 静态方法：staticmethod
3. 类方法：classmethod，第一个参数必须要默认传递，一般使用cls。

```
class Foo(object):
    """类三种方法语法形式"""
    def instance_method(self):
        print("是类{}的实例方法，只能被实例对象调用".format(Foo))

    @staticmethod
    def static_method():
        print("是静态方法")

    @classmethod
    def class_method(cls):
        print("是类方法")

# 创建实例
foo = Foo()
# 通过实例来调用方法
```

```
foo.instance_method()
foo.static_method()
foo.class_method()
print('----------------')
# 直接通过类，来调用实例方法会报错：缺少self
# Foo.instance_method()
# 直接通过类来调用方法
Foo.static_method()
Foo.class_method()
```

```
是类<class '__main__.Foo'>的实例方法，只能被实例对象调用
是静态方法
是类方法
----------------
是静态方法
是类方法
```

- 解释：
    1. 实例方法只能被实例调用，即：必须先进行实例化
    2. 静态方法/类方法，可以直接同类来调用

扩展

反射

```
class cl:
    country='China'
    def __init__(self,name,age):
        self.name=name
        self.age=age
obj=cl('Tom',22)
# 模拟用户输入，这是一个字符串
inp='name'
# 结果Tom
print(obj.name)
# 但是这样调用会报错 AttributeError: 'cl' object has no attribute 'inp'
# print(obj.inp)
# 使用getattr模拟：obj.inp 实际是：obj.name，可以防止报错
print(getattr(obj,inp,None))  #结果 noah
```

```
Tom
Tom
```

4大反射函数

1. hasattr(object,name) 判断object中有没有一个name字符串对应的方法或属性

2. getattr(object, name, default=None) 获取object中name字符串对应的方法或属性,如果不存在则返回None,可以定义其他返回值
3. setattr(object, name, values) 修改object中name属性为values值,name属性不存在,则新增改属性
4. delattr(object,name) 删除object对象中的name属性

```
# A同学只定义接口，但是未完成功能
class FtpClient:
    'ftp客户端,但是还么有实现具体的功能'
    def __init__(self,addr):
        print('正在连接服务器[%s]' %addr)
        self.addr=addr

    def get(self):
        pass

# B同学基于A同学的类，直接开发，而不用等待它开发完成
# from module import FtpClient
f1=FtpClient('192.168.1.1')
if hasattr(f1,'get'):
    func_get=getattr(f1,'get')
    func_get()
else:
    print('---->不存在此方法')
    print('处理其他的逻辑')
```

```
正在连接服务器[192.168.1.1]
```

```
```