18. 核能来袭-反射

本节主要内容:

1. isinstance, type, issubclass
2. 区分函数和方法
3. 反射(重点)

一. isinstance, type, issubclass

首先, 我们先看issubclass() 这个内置函数可以帮我们判断xxx类是否是yyy类型的子类.

```
class Base:
    pass
class Foo(Base):
    pass

class Bar(Foo):
    pass


print(issubclass(Bar, Foo))  # True
print(issubclass(Foo, Bar))  # False
print(issubclass(Bar, Base))  # True
```

然后我们来看type. type在前面的学习期间已经使用过了. type(obj) 表示查看obj是由哪个类创建的.

```
class Foo:
    pass

obj = Foo()
print(obj, type(obj))   # 查看obj的类
```

那这个鬼东西有什么用呢? 可以帮我们判断xxx是否是xxx数据类型的

```
class Boy:
    pass

class Girl:
    pass

# 统计传进来的男生和女生分别有多少
def func(*args):
    b = 0
    g = 0
    for obj in args:
```

```
        if type(obj) == Boy:
            b += 1
        elif type(obj) == Girl:
            g += 1
    return b, g

ret = func(Boy(), Girl(), Girl(), Girl(), Boy(), Boy(), Girl())

print(ret)
```

或者, 你在进行计算的时候. 先判断好要计算的数据类型必须是int或者float. 这样的计算才有意义.

```
def add(a, b):
    if (type(a) == int or type(a) == float) and (type(b) == int or type(b)
== float):
        return a + b
    else:
        print("我要报错")
```

isinstance也可以判断xxx是yyy类型的数据. 但是isinstance没有type那么精准.

```
class Base:
    pass

class Foo(Base):
    pass

class Bar(Foo):
    pass

print(isinstance(Foo(), Foo))   # True
print(isinstance(Foo(), Base))  # True

print(isinstance(Foo(), Bar))   # False
```

isinstance可以判断该对象是否是xxx家族体系中的(只能往上判断)

二. 区分函数和方法

我们之前讲过函数和方法. 这两样东西如何进行区分呢? 其实很简单. 我们只需要打印一下就能看到区别的.

```
def func():
    pass

print(func)     # <function func at 0x10646ee18>


class Foo:
```

```
    def chi(self):
        print("我是吃")


f = Foo()
print(f.chi)   # <bound method Foo.chi of <__main__.Foo object at
0x10f688550>>
```

函数在打印的时候. 很明显显示的是function. 而方法在打印的时候很明显是method. 那在这里. 我要告诉大家. 其实并不一定是这样的. 看下面的代码:

```
class Foo:

    def chi(self):
        print("我是吃")

    @staticmethod
    def static_method():
        pass

    @classmethod
    def class_method(cls):
        pass
f = Foo()
print(f.chi)   # <bound method Foo.chi of <__main__.Foo object at
0x10f688550>>
print(Foo.chi)  # <function Foo.chi at 0x10e24a488>
print(Foo.static_method)  # <function Foo.static_method at 0x10b5fe620>
print(Foo.class_method)  # bound method Foo.class_method of <class
'__main__.Foo'>>

print(f.static_method) # <function Foo.static_method at 0x10e1c0620>
print(f.class_method)   #<bound method Foo.class_method of <class
'__main__.Foo'>>
```

仔细观察, 我们能得到以下结论:

1. 类方法. 不论任何情况, 都是方法.
2. 静态方法, 不论任何情况. 都是函数
3. 实例方法, 如果是实例访问. 就是方法. 如果是类名访问就是函数.

那如何用程序来帮我们分辨. 到底是方法还是函数呢? 首先, 我们要借助于types模块.

```
# 所有的方法都是MethodType的实例
# 所有的函数都是FunctionType的实例
from types import MethodType, FunctionType
```

```
def func():
    pass

print(isinstance(func, FunctionType))   # True
print(isinstance(func, MethodType))     # False

class Foo:

    def chi(self):
        print("我是吃")

    @staticmethod
    def static_method():
        pass

    @classmethod
    def class_method(cls):
        pass

obj = Foo()
print(type(obj.chi))    # method
print(type(Foo.chi))    # function
print(isinstance(obj.chi, MethodType))  # True
print(isinstance(Foo.chi, FunctionType))    # True

print(isinstance(Foo.static_method, FunctionType))    # True
print(isinstance(Foo.static_method, MethodType))    # False

print(isinstance(Foo.class_method, FunctionType))    # False
print(isinstance(Foo.class_method, MethodType))    # True
```

用types中的FunctionType和MethodType可以区分, 当前内容是方法还是函数, 接下来. 看一个小题, 并分析答案

```
from types import FunctionType, MethodType

class Foo:
    @classmethod
    def func1(cls):
        pass

    @staticmethod
    def func2():
        pass

    def func3(self):
        pass

    def func4(self):
```

```
        pass

lst = [Foo.func1, Foo.func2, Foo.func3]
obj = Foo()
lst.append(obj.func4)

for item in lst:
    print(isinstance(item, MethodType))
    print(isinstance(item, FunctionType))
```

练习. 写一个函数. 判断传递进来的内容是函数还是方法?

三. 反射.

首先, 我们看这样一个需求, 说, 有个大牛, 写了一堆特别牛B的代码. 然后放在了一个py文件里(模块), 这时, 你想用这个大牛写的东西. 但是呢. 你首先得知道大牛写的这些代码都是干什么用的. 那就需要你把大牛写的每一个函数跑一下. 摘一摘自己想用的内容. 来咱们模拟这样的需求, 首先, 大牛给出一个模块.

大牛.py

```
def chi():
    print("大牛一顿吃100个螃蟹")

def he():
    print("大牛一顿喝100瓶可乐")

def la():
    print("大牛不用拉")

def shui():
    print("大牛一次睡一年")
```

接下来, 到你了. 你要去一个一个的调用. 但是呢. 在调用之前. 大牛告诉你了. 他写了哪些哪些方法. 那现在就可以这么来办了

```
import master

while 1:
    print("""作为大牛, 我帮你写了:
        chi
        he
        la
        shui
等功能. 自己看看吧""")
    gn = input("请输入你要测试的功能:")
    if gn == 'chi':
        master.chi()
    elif gn == "he":
```

```
        master.he()
    elif gn == "la":
        master.la()
    elif gn == "shui":
        master.shui()
    else:
        print("大牛就这几个功能. 别搞事情")
```

写是写完了. 但是.....如果大牛现在写了100个功能呢? 你的判断要判断100次么? 太累了吧. 现有的知识解决不了这个问题. 那怎么办呢? 注意看. 我们可以使用反射来完成这样的功能. 非常的简单. 想想. 这里我们是不是让用户输入要执行的功能了. 那这个功能就是对应模块里的功能. 那也就是说. 如果能通过字符串来动态访问模块中的功能就能解决这个问题. 好了. 我要告诉你. 反射解决的就是这个问题. 为什么叫反射? 反着来啊. 正常是我们先引入模块, 然后用模块去访问模块里的内容. 现在反了. 我们手动输入要运行的功能. 反着去模块里找. 这个就叫反射

```
import master

while 1:
    print("""作为大牛, 我帮你写了:
        chi
        he
        la
        shui
等功能. 自己看看吧""")
    gn = input("请输入你要测试的功能:")
    # niuB版
    func = getattr(master, gn)
    func()
```

getattr(对象, 字符串): 从对象中获取到xxx功能. 此时xxx是一个字符串. get表示找, attr表示属性(功能). 但是这里有个问题. 用户如果手一抖, 输入错了. 在大牛的代码里没有你要找的内容. 那这个时候就会报错. 所以. 我们在获取attr之前. 要先判断一下. 有没有这个attr.

完整代码:

```
import master
from types import FunctionType

while 1:
    print("""作为大牛, 我帮你写了:
        chi
        he
        la
        shui
等功能. 自己看看吧""")
    gn = input("请输入你要测试的功能:")
    # niuB版
    if hasattr(master, gn):  # 如果master里面有你要的功能
        # 获取这个功能, 并执行
```

```
        attr = getattr(master, gn)
        # 判断是否是函数. 只有函数才可以被调用
        if isinstance(attr, FunctionType):
            attr()
        else:
            # 如果不是函数, 就打印
            print(attr)
```

好了, 这里我们讲到了两个函数. 一个是getattr(). 一个是hasattr(). 其中getattr()用来获取信息. hasattr()用来判断xxx中是否包含了xxx功能, 那么我们可以在模块中这样来使用反射. 在面向对象中一样可以这样进行操作. 这个就比较牛B了. 后期你学习的相关框架内部核心源码几乎都是这些东西. 首先, 我们先看一些简单的.

```
class Person:

    country = "大清"
    def chi(self):
        pass


# 类中的内容可以这样动态的进行获取
print(getattr(Person, "country"))
print(getattr(Person, "chi"))    # 相当于Foo.func 函数

# 对象一样可以
obj = Person()
print(getattr(obj, "country"))
print(getattr(obj, "chi"))    # 相当于obj.func 方法
```

总结, getattr可以从模块中获取内容, 也可以从类中获取内容, 也可以从对象中获取内容. 在python中一切皆为对象. 那可以这样认为. getattr从对象中动态的获取成员

来看一个示例.

```
class Person:

    def chi(self):
        print("吃")

    def he(self):
        print("喝")

    def la(self):
        print("拉")

    def sa(self):
        print("撒")

    def shui(self):
        print("睡")
```

```
    def run(self):
        lst = ['chi', 'he', 'la', 'sa', 'shui']
        num = int(input("""本系统有以下功能
            1. 吃
            2. 喝
            3. 拉
            4. 撒
            5. 睡
请选择你要执行的功能:"""))

        # 通过类名也可以使用
        # func = getattr(Person, lst[num - 1])
        # func(self)

        # 通过对象来访问更加合理
        # method = getattr(self, lst[num-1])
        # method()

p = Person()
p.run()
```

补充:

关于反射, 其实一共有4个函数:

1. hasattr(obj, str) 判断obj中是否包含str成员
2. getattr(obj,str) 从obj中获取str成员
3. setattr(obj, str, value) 把obj中的str成员设置成value. 注意. 这里的value可以是值, 也可以是函数或者方法
4. delattr(obj, str) 把obj中的str成员删除掉

注意, 以上操作都是在内存中进行的. 并不会影响你的源代码

```
class Foo:
    pass


f = Foo()

print(hasattr(f, "chi"))    # False

setattr(f, "chi", "123")
print(f.chi)    # 被添加了一个属性信息


setattr(f, "chi", lambda x: x + 1)
print(f.chi(3)) # 4
```

```
print(f.chi) # 此时的chi既不是静态方法，也不是实例方法，更不是类方法. 就相当于你在类中写了个self.chi = lambda  是一样的
print(f.__dict__) # {'chi': <function <lambda> at 0x107f28e18>}


delattr(f, "chi")

print(hasattr(f, "chi"))    # False
```