反射

概述

运行时，区别于编译时，指的是程序被加载到内存中执行的时候。
反射，reflection，指的是运行时获取类型定义信息。
一个对象能够在运行时，像照镜子一样，反射出其类型信息。
简单说，在Python中，能够通过一个对象，找出其type、class、attribute或method的能力，称为反射或者自省。
具有反射能力的函数有：type()、isinstance()、callable()、dir()、getattr()

反射相关的函数和方法

需求
有一个Point类，查看它实例的属性，并修改它。动态为实例增加属性

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return "Point({}, {})".format(self.x, self.y)

    def show(self):
        print(self.x, self.y)

p = Point(4, 5)
print(p)
print(p.__dict__)
p.__dict__['y'] = 16
print(p.__dict__)
p.z = 10
print(p.__dict__)
print(dir(p)) # ordered list
print(p.__dir__()) # list
```

上例通过属性字典__dict__来访问对象的属性，本质上也是利用的反射的能力。

但是，上面的例子中，访问的方式不优雅，Python提供了内置的函数。

内建函数	意义
getattr(object, name[, default])	通过name返回object的属性值。当属性不存在，将使用default返回，如果没有default，则抛出AttributeError。name必须为字符串
setattr(object, name, value)	object的属性存在，则覆盖，不存在，新增
hasattr(object, name)	判断对象是否有这个名字的属性，name必须为字符串

用上面的方法来修改上例的代码

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return "Point({}, {})".format(self.x, self.y)

    def show(self):
        print(self)

p1 = Point(4, 5)
p2 = Point(10, 10)
print(repr(p1), repr(p2), sep='\n')
print(p1.__dict__)
setattr(p1, 'y', 16)
setattr(p1, 'z', 10)
print(getattr(p1, '__dict__'))

# 动态调用方法
if hasattr(p1, 'show'):
    getattr(p1, 'show')()


# 动态增加方法
# 为类增加方法
if not hasattr(Point, 'add'):
```

```
    setattr(Point, 'add', lambda self,other: Point(self.x + other.x, self.y + other
.y))

print(Point.add)
print(p1.add)
print(p1.add(p2)) # 绑定

# 为实例增加方法，未绑定
if not hasattr(p1, 'sub'):
    setattr(p1, 'sub', lambda self, other: Point(self.x - other.x, self.y - other.y
))

print(p1.sub(p1, p1))
print(p1.sub)

# add在谁里面，sub在谁里面
print(p1.__dict__)
print(Point.__dict__)
```

思考
这种动态增加属性的方式和装饰器修饰一个类、Mixin方式的差异？

这种动态增删属性的方式是运行时改变类或者实例的方式，但是装饰器或Mixin都是定义时就决定了，因此反射能力具有更大的灵活性。

练习
命令分发器，通过名称找对应的函数执行。
思路：名称找对象的方法

```
class Dispatcher:
    def __init__(self):
        self._run()

    def cmd1(self):
        print("I'm cmd1")

    def cmd2(self):
        print("I'm cmd2")

    def _run(self):
        while True:
```

```
            cmd = input('Plz input a command: ').strip()
            if cmd == 'quit':
                break
            getattr(self, cmd, lambda : print('Unknown Command {}'.format(cmd)))()

Dispatcher()
```

上例中使用getattr方法找到对象的属性的方式，比自己维护一个字典来建立名称和函数之间的关系的方式好多了。

反射相关的魔术方法

`__getattr__()` 、 `__setattr__()` 、 `__delattr__()` 这三个魔术方法，分别测试

`__getattr__()`

```
class Base:
    n = 0

class Point(Base):
    z = 6
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def show(self):
        print(self.x, self.y)

    def __getattr__(self, item):
        return "missing {}".format(item)

p1 = Point(4,5)
print(p1.x)
print(p1.z)
print(p1.n)
print(p1.t) # missing
```

一个类的属性会按照继承关系找，如果找不到，就会执行 `__getattr__()` 方法，如果没有这个

方法，就会抛出AttributeError异常表示找不到属性。

查找属性顺序为：

`instance.__dict__ --> instance.__class__.__dict__ --> 继承的祖先类（直到object）的__dict__ ---找不到--> 调用__getattr__()`

__setattr__()

```
class Base:
    n = 0

class Point(Base):
    z = 6
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def show(self):
        print(self.x, self.y)

    def __getattr__(self, item):
        return "missing {}".format(item)

    def __setattr__(self, key, value):
        print("setattr {}={}".format(key,value))

p1 = Point(4,5)
print(p1.x) # missing, why
print(p1.z)
print(p1.n)
print(p1.t) # missing
p1.x = 50
print(p1.__dict__)
p1.__dict__['x'] = 60
print(p1.__dict__)
print(p1.x)
```


实例通过.点设置属性，如同self.x = x，就会调用 `__setattr__()` ，属性要加到实例的 `__dict__` 中，就需要自己完成。

```
class Point(Base):
    z = 6
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def show(self):
        print(self.x, self.y)

    def __getattr__(self, item):
        return "missing {}".format(item)

    def __setattr__(self, key, value):
        print("setattr {}={}".format(key,value))
        self.__dict__[key] = value
```

`__setattr__()` 方法，可以拦截对实例属性的增加、修改操作，如果要设置生效，需要自己操作实例的 `__dict__` 。

`__delattr__()`

```
class Point:
    Z = 5
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __delattr__(self, item):
        print('Can not del {}'.format(item))

p = Point(14, 5)
del p.x
p.z = 15
del p.z
del p.Z
print(Point.__dict__)
print(p.__dict__)
del Point.Z
print(Point.__dict__)
```

可以阻止通过实例删除属性的操作。但是通过类依然可以删除属性。

__getattribute__

```
class Base:
    n = 0

class Point(Base):
    z = 6
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __getattr__(self, item):
        return "missing {}".format(item)

    def __getattribute__(self, item):
        return item


p1 = Point(4,5)
print(p1.__dict__)
print(p1.x)
print(p1.z)
print(p1.n)
print(p1.t)
print(Point.__dict__)
print(Point.z)
```

实例的所有的属性访问，第一个都会调用 `__getattribute__` 方法，它阻止了属性的查找，该方法应该返回（计算后的）值或者抛出一个AttributeError异常。
它的return值将作为属性查找的结果。如果抛出AttributeError异常，则会直接调用 `__getattr__` 方法，因为表示属性没有找到。

```
class Base:
    n = 0

class Point(Base):
    z = 6
```

```
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __getattr__(self, item):
        return "missing {}".format(item)

    def __getattribute__(self, item):
        #raise AttributeError("Not Found")
        #pass
        #return self.__dict__[item]
        return object.__getattribute__(self, item)

p1 = Point(4,5)
print(p1.__dict__)
print(p1.x)
print(p1.z)
print(p1.n)
print(p1.t)
print(Point.__dict__)
print(Point.z)
```


`__getattribute__` 方法中为了避免在该方法中无限的递归，它的实现应该永远调用基类的同名方法以访问需要的任何属性，例如 `object.__getattribute__(self, name)` 。
注意，除非你明确地知道 `__getattribute__` 方法用来做什么，否则不要使用它。

总结

魔术方法	意义
`__getattr__()`	当通过搜索实例、实例的类及祖先类查不到属性，就会调用此方法
`__setattr__()`	通过.访问实例属性，进行增加、修改都要调用它
`__delattr__()`	当通过实例来删除属性时调用此方法
`__getattribute__`	实例所有的属性调用都从这个方法开始

属性查找顺序：

`实例调用__getattribute__() --> instance.__dict__ --> instance.__class__.__dict__ --> 继承的祖先类（直到object）的__dict__ --> 调用__getattr__()`