小白必备面向对象练习

68 Python 中如何定义一个自己的 class？

Python 使用关键字 class 定制自己的类，self 表示类实例对象本身。

一个自定义类内包括属性、方法，其中有些方法是自带的。

类（对象）：

复制

```
class Dog(object):
    pass
```

以上定义一个 Dog 对象，它继承于根类 object，pass 表示没有自定义任何属性和方法。

69 classmethod 使用总结

classmethod 修饰符对应的函数不需要实例化，不需要 self 参数。

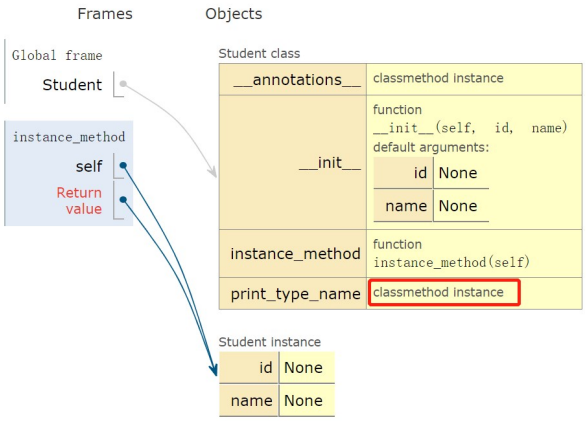
第一个参数需要是表示自身类的 cls 参数，能调用类的属性、方法、实例等。

复制

```
In [77]: class Student():
    ...:     def __init__(self,id=None,name=None):
    ...:         self.id = id
    ...:         self.name = name
    ...:
    ...:     def instance_method(self):
    ...:         print('这是实例方法')
    ...:         return self
    ...:
    ...:     @classmethod
    ...:     def __annotations__(cls):
    ...:         return "学生类"
    ...:
    ...:     @classmethod
    ...:     def print_type_name(cls):
    ...:         print('这是类上的方法，类名为 %s，注解为 %s'%(cls.__name__, cl
s.__annotat
    ...: ions__()))
```

复制

```
In [78]: Student().print_type_name()
    ...: Student().instance_method()
    ...:
这是类上的方法，类名为 Student，注解为 学生类
这是实例方法
Out[78]: <__main__.Student at 0x154ef861308>
```



70 删除对象的属性

delattr(object, name)

删除对象的属性，在不需要某个或某些属性时，这个方法就会很有用。

复制

```
In [79]: class Student():
    ...:     def __init__(self,id=None,name=None):
    ...:         self.id = id
    ...:         self.name = name

In [80]: xiaoming = Student(1,'xiaoming')

In [81]: delattr(xiaoming,'id')

In [82]: xiaoming.id
AttributeError: 'Student' object has no attribute 'id'

In [88]: hasattr(xiaoming,'id') # xiaoming上没有 id 属性
Out[88]: False
```

71 获取对象的属性

getattr(object, name[, default])

获取对象的属性

复制

```
In [79]: class Student():
    ...:     def __init__(self,id=None,name=None):
    ...:         self.id = id
    ...:         self.name = name

In [106]: getattr(xiaoming,'name')
Out[106]: 'xiaoming'
```

72 判断对象是否有某个属性名称

hasattr(object, name)

```
In [79]: class Student():
    ...:     def __init__(self,id=None,name=None):
    ...:         self.id = id
    ...:         self.name = name

In [110]: hasattr(xiaoming,'name')
Out[110]: True

In [81]: delattr(xiaoming,'id')

In [111]: hasattr(xiaoming,'id')
Out[111]: False
```

73 实例属于某个对象判断

isinstance(object, classinfo)

判断 `object` 是否为类 `classinfo` 的实例，若是，返回true

```
In [79]: class Student():
    ...:     def __init__(self,id=None,name=None):
    ...:         self.id = id
    ...:         self.name = name

In [21]: xiaoming = Student('001','xiaoming')
In [22]: isinstance(xiaoming,Student)
Out[22]: True
```

序列类型的基类为 `Iterable` ，所以返回 `True`

```
In [85]: from collections.abc import Iterable

In [84]: isinstance([1,2,3],Iterable)
Out[84]: True
```

74 子类判断

issubclass(class, classinfo)

如果 class 是 classinfo 类的子类，返回 `True` ：

```
In [27]: class undergraduate(Student):
    ...:     def studyClass(self):
    ...:         pass
    ...:     def attendActivity(self):
    ...:         pass
    ...:

In [28]: issubclass(undergraduate,Student)
Out[28]: True

In [29]: issubclass(object,Student)
Out[29]: False

In [30]: issubclass(Student,object)
Out[30]: True
```

classinfo 取值也可能为元组，若 class 是元组内某个元素类型的子类，也会返回 True

```
In [26]: issubclass(int, (int,float) )
Out[26]: True
```

75 鸭子类型

Python 是动态语言，对函数参数的类型要求很宽松，函数体内使用此类型的方法或属性时，只要满足有它们就行，不强制要求必须为这个类或子类。但是，对静态类型语言，如 `Java` ，参数类型就必须为此类型或子类。

例如，下面定义一个 `Plane` 类，定义函数 `using_run` ：

```
class Plane():
    def run(self):
        print('plane is flying...')

def using_run(duck):
    print(duck.run())

using_run(Plane())
```

打印结果：

```
plane is flying...
```

定义一个 `Clock` 类，它与 `Plane` 类没有继承关系，但是也有一个 `run` 方法：

```
class Clock():
    def run(self):
        print('clock is rotating...')
```

`using_run` 函数中，同样可传入 `Clock` 对象：

```
using_run(Clock())
```

打印结果：

```
clock is rotating...
```

`Plane` 对象和 `Clock` 对象，因都有 `run` 方法， Python 认为它们看起来就是 duck 类型，因此， `Plane` 对象和 `Clock` 对象就被看作 duck 类型。

76 有什么方法获取类的所有属性和方法？

获取下面类 Student 的所有属性和方法，使用 `dir()` 内置函数。

```
class Student:
    def __init__(self,id,name):
        self.id = id
        self.name = name
    def __eq__(self,student):
        return self.id == student.id
```

获取类上的所有属性和方法

```
In [87]: dir(Student)
Out[87]:
['class',
 'delattr',
 'dict',
 'dir',
 'doc',
 'eq',
 'format',
 'ge',
 'getattribute',
 'gt',
 'hash',
 'init',
 'init_subclass',
 'le',
 'lt',
 'module',
 'ne',
 'new',
 'reduce',
 'reduce_ex',
 'repr',
 'setattr',
 'sizeof',
 'str',
 'subclasshook',
 'weakref']
```

获取实例上的属性和方法：

```
s1 = Student(10,'xiaoming')
In [88]: dir(s1)
Out[88]:
['class',
 'delattr',
 'dict',
 'dir',
 'doc',
 'eq',
 'format',
 'ge',
 'getattribute',
 'gt',
 'hash',
 'init',
 'init_subclass',
 'le',
 'lt',
 'module',
 'ne',
 'new',
 'reduce',
 'reduce_ex',
 'repr',
 'setattr',
 'sizeof',
 'str',
 'subclasshook',
 'weakref',
 'id',
 'name']
```

77 Python 中如何动态获取和设置对象的属性？

如下 Student 类：

```
class Student:
    def __init__(self,id,name):
        self.id = id
        self.name = name
    def __eq__(self,student):
        return self.id == student.id
```

Python 使用 `hasattr` 方法，判断实例是否有属性 x：

```
s1 = Student(10,'xiaoming')
In [93]: hasattr(s1,'id')
Out[93]: True
In [94]: hasattr(s1,'address')
Out[94]: False
```

使用 `setattr` 动态添加对象的属性，函数原型：

```
<function setattr(obj, name, value, /)>
```

为类对象 `Student` 添加属性：

```
if not hasattr(Student, 'address'):
    setattr(Student,'address','beijing')
    print(hasattr(s1,'address'))
```

78 面向对象设计中 super 如何使用？

super([type[, object-or-type]])

返回一个代理对象，它会将方法调用委托给 type 的父类或兄弟类。

如下，子类的 `add` 方法，一部分直接调用父类 `add` ，再有一部分个性的行为：打印结果。

```
In [1]: class Parent():
```

```
    ...:     def __init__(self,x):
    ...:         self.v = x
    ...:
    ...:     def add(self,x):
    ...:         return self.v + x

In [2]: class Son(Parent):
    ...:     def add(self,y):
    ...:         r = super().add(y) #直接调用父类的add方法
    ...:         print(r) #子类的add与父类相比，能实现对结果的打印功能
    ...:

In [3]: Son(1).add(2)
3
```

79 判断对象是否可被调用之callable函数

callable(object)

判断对象是否可被调用，能被调用的对象就是一个`callable`对象，比如函数`str`，`int`等都是可被调用的。

复制

```
In [1]: callable(str)
Out[1]: True

In [2]: callable(int)
Out[2]: True
```

如下，`xiaoming`实例不可被调用：

复制

```
In [79]: class Student():
    ...:     def __init__(self,id=None,name=None):
    ...:         self.id = id
    ...:         self.name = name

In [21]: xiaoming = Student('001','xiaoming')

In [4]: callable(xiaoming)
Out[4]: False
```

如果`xiaoming`能被调用：

复制

```
xiaoming()
```

必须要重写`Student`类上`__call__`方法：

复制

```
In [1]: class Student():
    ...:     def __init__(self,id,name):
    ...:         self.id = id
    ...:         self.name = name
    ...:     def __call__(self):
    ...:         print('I can be called')
    ...:         print(f'my name is {self.name}')

In [2]: t = Student('001','xiaoming')

In [3]: t()
I can be called
my name is xiaoming
```

下一章