

小强识别关键字练习

190 @property使用解释说明

```
property(fget=None, fset=None, fdel=None, doc=None)
```

返回 property 属性

不适用装饰器，定义类上的属性：

```
class Student:
    def __init__(self):
        self._name = None

    def get_name(self):
        return self._name

    def set_name(self, val):
        self._name = val

    def del_name(self):
        del self._name

# 显示调用 property 函数
name = property(get_name, set_name, del_name, "name property")
```

[复制](#)

显示的调用 property 函数定义类上的属性：

```
In [94]: xiaoming = Student()
...: xiaoming.name = 'xiaoming'

In [95]: xiaoming.name
Out[95]: 'xiaoming'
```

[复制](#)

使用 python 装饰器 @property，同样能实现对类上属性的定义，并且更简洁：

```
class Student:
    def __init__(self):
        self._name = None

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, val):
        self._name = val

    @name.deleter
    def name(self):
        del self._name
```

[复制](#)

```
In [98]: xiaoming = Student()

In [99]: xiaoming.name = 'xiaoming'

In [101]: xiaoming.name
Out[101]: 'xiaoming'
```

[复制](#)

191 标识符 is 作用

is 判断标识号是否相等

is 比较的是两个对象的标识号是否相等，Python 中使用 id() 函数获取对象的标识号。

```
In [1]: a = [1,2,3]
In [2]: id(a)
Out[2]: 95219592

In [5]: b = [1,2,3] #再创建一个列表实例，元素取值也为 1,2,3
In [6]: id(b)
Out[6]: 95165640
```

[复制](#)

创建的两个列表实例位于不同的内存地址，所以它们的标识号不等。

```
In [7]: a is b
Out[7]: False
```

[复制](#)

即便对于两个空列表实例，它们 is 比较的结果也是 False

192 标识符 is 的特殊之处

对于值类型而言，不同的编译器可能会做不同的优化。

从性能角度考虑，它们会缓存一些值类型的对象实例。

所以，使用 is 比较时，返回的结果看起来会有些不太符合预期。

注意观察下面两种情况，同样的整数值，使用 is 得到不同结果。

```
In [100]: a = 123
In [101]: b = 123
In [102]: a is b
Out[102]: True
```

[复制](#)

```
In [103]: c = 123456
In [104]: d = 123456

In [105]: c is d
Out[105]: False
```

[复制](#)

Python解释器，对位于区间 [-5, 256] 内的小整数，会进行缓存，不在该范围内的不会缓存，所以才出现上面的现象。

[190 @property使...](#)[191 标识符 is 作用](#)[192 标识符 is 的特...](#)[193 标识符 in 的用法](#)[194 对于自定义类...](#)[195 标识符 == 的...](#)[196 标识符 == 的...](#)[197 is和==有什么...](#)[198 nonlocal关键...](#)[199 global 关键字...](#)[200 lambda 函数的...](#)

in 用于成员检测

如果元素 `i` 是 `s` 的成员，则 `i in s` 为 `True`

若不是 `s` 的成员，则返回 `False`，也就是 `i not in s` 为 `True`

对于字符串类型，`i in s` 为 `True`，意味着 `i` 是 `s` 的子串，也就是 `s.find(i)` 返回大于 `-1` 的值。举例如下：

```
In [70]: 'ab' in 'abc'
Out[70]: True
In [27]: 'abc'.find('ab')
Out[27]: 0

In [71]: 'ab' in 'acb'
Out[71]: False
In [29]: 'abc'.find('ac')
Out[29]: -1
```

内置的序列类型、字典类型和集合类型，都支持 `in` 操作。

对于字典类型，`in` 操作判断 `i` 是否是字典的键。

```
In [30]: [1,2] in [[1,2], 'str']
Out[30]: True

In [31]: 'apple' in {'orange':1.5, 'banana':2.3, 'apple':5.2}
Out[31]: True
```

194 对于自定义类型，判断成员是否位于序列类型中的方法

对于自定义类型，判断是否位于序列类型中，需要重写序列类型的魔法方法 `__contains__`。

具体操作步骤如下：

- 自定义 `Student` 类，无特殊之处
- `Students` 类继承 `list`，并重写 `__contains__` 方法

根据 `Student` 类的 `name` 属性，判断某 `Student` 是否在 `Students` 序列对象中。

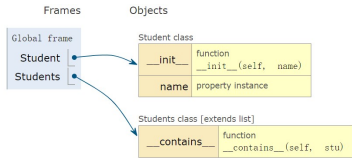
```
class Student():
    def __init__(self, name):
        self._name = name

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, val):
        self._name = val

class Students(list):
    def __contains__(self, stu):
        for s in self:
            if s.name == stu.name:
                return True
        return False
```

`Student`，`Students` 类的示意图：



使用自定义类，`s3` 的名字与列表 `a` 中的第一个元素 `s1` 重名，所以 `s3 in a` 返回 `True`。

`s4` 不在列表 `a` 中，所以 `in` 返回 `False`。

```
s1 = Student('xiaoming')
s2 = Student('xiaohong')

a = Students()
a.extend([s1,s2])

s3 = Student('xiaoming')
print(s3 in a) # True

s4 = Student('xiaoli')
print(s4 in a) # False
```

195 标识符 `==` 的作用

`==` 判断值是否相等

对于数值型、字符串、列表、字典、集合，默认只要元素值相等，`==` 比较结果是 `True`。

如下所示：

```
In [46]: str1 = "alg-channel"

In [47]: str2 = "alg-channel"

In [48]: str1 == str2
Out[48]: True

In [49]: a = [1, 2, 3]

In [50]: b = [1, 2, 3]

In [51]: a==b
Out[51]: True
```

```

In [52]: c = [1, 3, 2]

In [53]: a==c
Out[53]: False

In [54]: a = {'a':1.0,'b':2.0}

In [55]: b = {'a':1.0,'b':2.0}

In [56]: a==b
Out[56]: True

In [57]: c = (1,2)

In [58]: d = (1,2)

In [59]: c==d
Out[59]: True

In [60]: c={1,2,3}

In [61]: d={1,3,2}

In [62]: c==d
Out[62]: True

```

196 标识符 == 的实际应用场景

对于自定义类型，当所有属性取值完全相同的两个实例，判断 == 时，返回 False。

但是，大部分场景下，我们希望这两个对象是相等的，这样不用重复添加到列表中。

比如，判断用户是否已经登入时，只要用户所有属性与登入列表中某个用户完全一致时，就认为已经登入。

如下所示，需要重写方法 __eq__，使用 __dict__ 获取实例的所有属性。

```

class Student():
    def __init__(self,name,age):
        self._name = name
        self._age = age

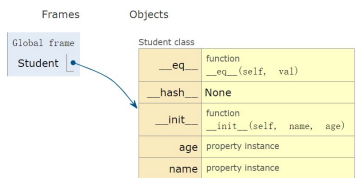
    @property
    def name(self):
        return self._name
    @name.setter
    def name(self,val):
        self._name = val

    @property
    def age(self):
        return self._age
    @age.setter
    def age(self,val):
        self._age = val

    def __eq__(self,val):
        print(self.__dict__)
        return self.__dict__ == val.__dict__

```

Student 类的示意图：



如下，第三个实例 xiaoming2 与 已添加到列表 a 中的 xiaoming 属性完全一致，所以 == 比较或 in 时，都会返回 True，这正是我们想要的结果。

```

a = []
xiaoming = Student('xiaoming',29)
if xiaoming not in a:
    a.append(xiaoming)

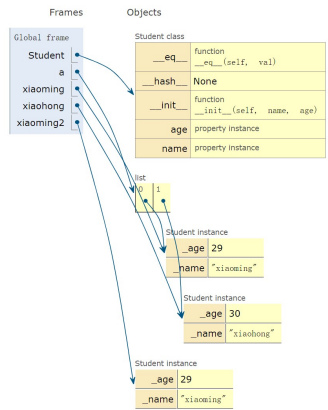
xiaohong = Student('xiaohong',30)
if xiaohong not in a:
    a.append(xiaohong)

xiaoming2 = Student('xiaoming',29)
if xiaoming2 == xiaoming:
    print('对象完全一致.相等')

if xiaoming2 not in a:
    a.append(xiaoming2)

print(len(a))

```



197 is和==有什么区别？

`i` 用来判断两个对象的标识号是否相等；

`==` 用于判断值或内容是否相等，默认是基于两个对象的标识号比较。

也就是说，如果 `a` 是 `b` 为 `True` 且如果按照默认行为，意味着 `a==b` 也为 `True`

198 nonlocal关键字及使用案例

关键词 `nonlocal` 常用于函数嵌套中，声明某个变量为**非局部变量**。

下面案例，说明 `nonlocal` 的功能。

如下，函数 `f` 里嵌套一个函数 `auto_increase`。实现功能：不大于 10 时自增，否则置零后，再从零自增。

```
In [4]: def f():
...:     i=0
...:     def auto_increase():
...:         if i>=10:
...:             i = 0
...:             i+=1
...:     ret = []
...:     for _ in range(28):
...:         auto_increase()
...:         ret.append(i)
...:     print(ret)
```

调用函数 `f`，会报出如下异常：

```
<ipython-input-9-5ca6794fdb70> in auto_increase()
      2     i=0
      3     def auto_increase():
----> 4         if i>=10:
          i = 0
          i+=1
UnboundLocalError: local variable 'i' referenced before assignment
```

`if i>=10` 这行报错，`i` 引用前未被赋值。

为什么会这样？明明 `i` 一开始已经被定义！

原来，最靠近变量 `i` 的函数是 `auto_increase`，不是 `f`，`i` 没有在 `auto_increase` 中先赋值，所以报错。

解决方法：使用 `nonlocal` 声明 `i` 不是 `auto_increase` 内的局部变量。

修改方法：

```
In [11]: def f():
...:     i=0
...:     def auto_increase():
...:         nonlocal i # 使用 nonlocal 告诉编译器, i 不是局部变量
...:         if i>=10:
...:             i = 0
...:             i+=1
...:     ret = []
...:     for _ in range(28):
...:         auto_increase()
...:         ret.append(i)
...:     print(ret)

In [12]: f()
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1, 2, 3, 4, 5, 6, 7, 8]
```

调用 `f()`，正常输出结果。

199 global 关键字用法及使用案例

先回答为什么要有 `global`：一个变量被多个函数引用，想让全局变量被所有函数共享。

有的读者可能会觉得这不简单，这样写：

```
i = 5
def f():
    print(i)

def g():
    print(i)
    pass

f()
g()
```

`f` 和 `g` 两个函数都能共享变量 `i`，程序没有报错。所以，至此依然没有真正解释 `global` 存在的价值。

但是，如果某个函数要修改 `i`，实现递增，这样：

```
def h():
    i += 1
    h()
```

此时执行程序，就会出错，抛出异常：`UnboundLocalError`。

原来，编译器在解释 `i+=1` 时，会解析 `i` 为函数 `h()` 内的局部变量。

很显然，在此函数内，解释器找不到对变量 `i` 的定义，所以报错。

`global` 在此种场景下，会大显身手。

在函数 `h` 内，显示地告诉解释器 `i` 为全局变量，然后，解释器会在函数外面寻找 `i` 的定义，执行完 `i+=1` 后，`i` 还为全局变量，值加 1：

```
i = 0
def h():
    global i
```

```
i += 1

h()
print(i)
```

200 lambda 函数的形参和返回值

key 值为 lambda 函数，说说 lambda 函数的形参和返回值

```
def longer(*s):
    return max(*s, key=lambda x: len(x))

longer({1,3,5,7},{1,5,7},{2,4,6,7,8})
```

lambda 函数的形参：s 解包后的元素值，


可能取值为：{1,3,5,7}, {1,5,7}, {2,4,6,7,8} 三种；

lambda 函数的返回值为：元素的长度，

可能取值为：{1,3,5,7}, {1,5,7}, {2,4,6,7,8} 的长度4,3,5


下一章

互动评论




说点什么

评论




The Scrapper


200道题了，不错

 鼓掌

1个月前



存

 1

<

>