**全面了解Python类的内置方法**

**Python中的类提供了很多双下划线开头和结尾\_\_xxx\_\_的方法，这些方法是Python运行的基础，很多功能背后都是通过调用这些内置方法来实现的。一起来了解一下吧！**

**1、new、init**

\_\_new\_\_方法是真正的类构造方法，用于产生实例化对象（空属性）。重写\_\_new\_\_方法可以控制对象的产生过程。  
\_\_init\_\_方法是初始化方法，负责对实例化对象进行属性值初始化，此方法必须返回None，\_\_new\_\_方法必须返回一个对象。重写\_\_init\_\_方法可以控制对象的初始化过程。

**使用new来处理单例模式：**

**class** **Student**:

\_\_instance **=** **None**

**def** **\_\_new\_\_**(cls, **\***args, **\*\***kwargs):

**if** **not** cls**.**\_\_instance:

cls**.**\_\_instance **=** object**.**\_\_new\_\_(cls)

**return** cls**.**\_\_instance

**def** **sleep**(self):

print('sleeping...')

stu1 **=** Student()

stu2 **=** Student()

print(id(stu1), id(stu2)) *# 两者输出相同*

print(stu1 **is** stu2) *# True*

\_\_new\_\_一般很少用于普通的业务场景，更多的用于元类之中，因为可以更底层的处理对象的产生过程。而\_\_init\_\_的使用场景更多。

**2、str、repr**

两者的目的都是为了显式的显示对象的一些必要信息，方便查看和调试。\_\_str\_\_被print默认调用，\_\_repr\_\_被控制台输出时默认调用。即，使用\_\_str\_\_控制用户展示，使用\_\_repr\_\_控制调试展示。

**使用\_*str*\_控制对象的输出：**

# 自定义str来控制print的显示内容，str函数必须return一个字符串对象

# 使用repr = str来偷懒控制台和print的显示一致

class Student:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

def \_\_str\_\_(self):

return f'{self.\_\_class\_\_}, {self.name}, {self.age}'

\_\_repr\_\_ = \_\_str\_\_

stu = Student('zlw', 26)

print(stu) # <class '\_\_main\_\_.Student'>, zlw, 26

**3、call**

\_\_call\_\_方法提供给对象可以被执行的能力，就像函数那样，而本质上，函数就是对象，函数就是一个拥有\_\_call\_\_方法的对象。拥有\_\_call\_\_方法的对象，使用callable可以得到True的结果，可以使用（）执行，执行时，可以传入参数，也可以返回值。所以我们可以使用\_\_call\_\_方法来实现实例化对象作为装饰器：

**用\_*call*\_来实现类装饰器**

# 检查一个函数的输入参数个数,

# 如果调用此函数时提供的参数个数不符合预定义，则无法调用。

# 实例对象版本装饰器

class Checker:

def \_\_init\_\_(self, require\_num):

self.require\_num = require\_num

def \_\_call\_\_(self, func):

self.func = func

def inner(\*args, \*\*kw):

if len(args) != self.require\_num:

print('函数参数个数不符合预定义，无法执行函数')

return None

return self.func(\*args, \*\*kwargs)

return inner

@Checker(2)

def show(\*args):

print('show函数成功执行!')

show(1) # 函数参数个数不符合预定义，无法执行函数

show(1,2) # show函数成功执行!

show(1,2,3) # 函数参数个数不符合预定义，无法执行函数

**4、del**

\_\_del\_\_用于当对象的引用计数为0时自动调用。  
\_\_del\_\_一般出现在两个地方：1、手工使用del减少对象引用计数至0，被垃圾回收处理时调用。2、程序结束时调用。  
\_\_del\_\_一般用于需要声明在对象被删除前需要处理的资源回收操作

# 手工调用del 可以将对象引用计数减一，如果减到0，将会触发垃圾回收

class Student:

def \_\_del\_\_(self):

print('调用对象的del方法，此方法将会回收此对象内存地址')

stu = Student() # 调用对象的\_\_del\_\_方法回收此对象内存地址

del stu

print('下面还有程序其他代码')

程序直接结束，也会调用对象的\_\_del\_\_方法回收地址：

class Student:

def \_\_del\_\_(self):

print('调用对象的del方法，此方法将会回收此对象内存地址')

stu = Student() # 程序直接结束，也会调用对象的\_\_del\_\_方法回收地址

**5、iter、next**

这2个方法用于将一个对象模拟成**序列**。内置类型如列表、元组都可以被迭代，文件对象也可以被迭代获取每一行内容。重写这两个方法就可以实现自定义的迭代对象。

# 定义一个指定范围的自然数类，并可以提供迭代

class Num:

def \_\_init\_\_(self, max\_num):

self.max\_num = max\_num

self.count = 0

def \_\_iter\_\_(self):

return self

def \_\_next\_\_(self):

if self.count < self.max\_num:

self.count += 1

return self.count

else:

raise StopIteration('已经到达临界')

num = Num(10)

for i in num:

print(i) # 循环打印1---10

**6、getitem、setitem、delitem**

重写此系列方法可以模拟对象成列表或者是字典，即可以使用key-value的类型。

class StudentManager:

li = []

dic = {}

def add(self, obj):

self.li.append(obj)

self.dic[obj.name] = obj

def \_\_getitem\_\_(self, item):

if isinstance(item, int):

# 通过下标得到对象

return self.li[item]

elif isinstance(item, slice):

# 通过切片得到一串对象

start = item.start

stop = item.stop

return [student for student in self.li[start:stop]]

elif isinstance(item, str):

# 通过名字得到对象

return self.dic.get(item, None)

else:

# 给定的key类型错误

raise TypeError('你输入的key类型错误!')

class Student:

manager = StudentManager()

def \_\_init\_\_(self, name):

self.name = name

self.manager.add(self)

def \_\_str\_\_(self):

return f'学生: {self.name}'

\_\_repr\_\_ = \_\_str\_\_

stu1 = Student('小明')

stu2 = Student('大白')

stu3 = Student('小红')

stu4 = Student('胖虎')

# 当做列表使用

print(Student.manager[0]) # 学生: 小明

print(Student.manager[-1]) # 学生: 胖虎

print(Student.manager[1:3]) # [学生: 大白, 学生: 小红]

# 当做字典使用

print(Student.manager['胖虎']) # 学生: 胖虎

**7、getattr、setattr、delattr**

当使用obj.x = y的时候触发对象的setattr方法，当del obj.x的时候触发对象的delattr方法。  
当尝试访问对象的一个不存在的属性时 obj.noexist 会触发getattr方法，getattr方法是属性查找中优先级最低的。  
可以重写这3个方法来控制对象属性的访问、设置和删除。  
**特别注意：如果定义了getattr，而没有任何代码（即只有pass），则所有不存在的属性值都是None而不会报错,可以使用super().getattr()方法来处理**

class Student:

def \_\_getattr\_\_(self, item):

print('访问一个不存在的属性时候触发')

return '不存在'

def \_\_setattr\_\_(self, key, value):

print('设置一个属性值的时候触发')

# self.key = value # 这样会无限循环

self.\_\_dict\_\_[key] = value

def \_\_delattr\_\_(self, item):

print('删除一个属性的时候触发')

if self.\_\_dict\_\_.get(item, None):

del self.\_\_dict\_\_[item]

stu = Student()

stu.name = 'zlw' # 设置一个属性值的时候触发

print(stu.noexit) # 访问一个不存在的属性时候触发 , 返回'不存在'

del stu.name # 删除一个属性的时候触发

**8、getatrribute**

这是一个**属性访问截断器**，即，在你访问属性时，这个方法会把你的访问行为截断，并优先执行此方法中的代码，此方法应该是属性查找顺序中优先级最高的。  
**属性查找顺序：**  
**实例的getattribute-->实例对象字典-->实例所在类字典-->实例所在类的父类(MRO顺序）字典-->实例所在类的getattr-->报错**

class People:

a = 200

class Student(People):

a = 100

def \_\_init\_\_(self, a):

self.a = a

def \_\_getattr\_\_(self, item):

print('没有找到:', item)

def \_\_getattribute\_\_(self, item):

print('属性访问截断器')

if item == 'a':

return 1

return super().\_\_getattribute\_\_(item)

stu = Student(1)

print(stu.a) # 输出结果为：1

**9、enter、exit**

这两个方法的重写可以让我们对一个对象使用with方法来处理工作前的准备，以及工作之后的清扫行为。

下面使用\_\_enter\_\_() 和 \_\_exit\_\_() 实现了一个上下文管理器：

class MySQL:

def connect(self):

print('启动数据库连接，申请系统资源')

def execute(self):

print('执行sql命令，操作数据')

def finish(self):

print('数据库连接关闭，清理系统资源')

def \_\_enter\_\_(self): # with的时候触发，并赋给as变量

self.connect()

return self

def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb): # 离开with语句块时触发

self.finish()

with MySQL() as mysql:

mysql.execute()

# 结果:

# 启动数据库连接，申请系统资源

# 执行sql命令，操作数据

# 数据库连接关闭，清理系统资源

**10、get、set、delete描述器**

一般而言，描述器是一个包含了描述器协议中的方法的属性值。这些方法是 \_\_get\_\_()，\_\_set\_\_() 和 \_\_delete\_\_()。如果某个属性定义了这些方法中的任意一个，那么就可以说它是一个 descriptor。

属性访问的默认行为是从一个对象的字典中获取、设置或删除属性。对于实例来说，a.x 的查找顺序会从 a.\_\_dict\_\_['x'] 开始，然后是 type(a).\_\_dict\_\_['x']，接下来依次查找 type(a) 的方法解析顺序（MRO）。 如果找到的值是定义了某个描述器方法的对象，则 Python 可能会重写默认行为并转而发起调用描述器方法。这具体发生在优先级链的哪个环节则要根据所定义的描述器方法及其被调用的方式来决定。

描述器是一个强大而通用的协议。 它们是属性、方法、静态方法、类方法和 super() 背后的实现机制。 它们在 Python 内部被广泛使用。 描述器简化了底层的 C 代码并为 Python 的日常程序提供了一组灵活的新工具。

**描述符协议**

* descr.\_\_get\_\_(self, obj, type=None) -> value
* descr.\_\_set\_\_(self, obj, value) -> None
* descr.\_\_delete\_\_(self, obj) -> None

定义任何上面三个方法的任意一个,这个对象就会被认为是一个描述符,并且可以在被作为对象属性时重载默认的行为, 如果一个对象定义了\_\_get\_\_() 和 \_\_set\_\_(),它被认为是一个数据描述符.只定义 \_\_get\_\_()被认为是非数据描述符,数据和非数据描述符的区别在于:如果一个实例的字典有和数据描述符同名的属性,那么数据描述符会被优先使用,如果一个实例的字典实现了无数据描述符的定义,那么这个字典中的属性会被优先使用,实现只读数据描述符,同时定义\_\_get\_\_()和\_\_set\_\_(),在\_\_set\_\_()中抛出AttributeError.

如果一个对象定义了 \_\_set\_\_() 或 \_\_delete\_\_()，则它会被视为数据描述器。 仅定义了 \_\_get\_\_() 的描述器称为非数据描述器（它们经常被用于方法，但也可以有其他用途）。

数据和非数据描述器的不同之处在于，如何计算实例字典中条目的替代值。如果实例的字典具有与数据描述器同名的条目，则数据描述器优先。如果实例的字典具有与非数据描述器同名的条目，则该字典条目优先。

为了使数据描述器成为只读的，应该同时定义 \_\_get\_\_() 和 \_\_set\_\_() ，并在 \_\_set\_\_() 中引发 AttributeError 。用引发异常的占位符定义 \_\_set\_\_() 方法使其成为数据描述器。

# 创建一个描述器的类，它的实例就是一个描述器

# 这个类要有\_\_get\_\_ \_\_set\_\_ 这样的方法

# 这种类是当做工具使用的，不单独使用

class M:

def \_\_init\_\_(self, x=1):

self.x = x

def \_\_get\_\_(self, instance, owner):

return self.x

def \_\_set\_\_(self, instance, value):

self.x = value

# 调用描述器的类

class AA:

m = M() # m就是一个描述器

aa = AA()

aa.m # 1

aa.m = 2

aa.m # 2