**Python中下划线---完全解读**

[Python](http://lib.csdn.net/base/python) 用下划线作为变量前缀和后缀指定特殊变量

\_xxx 不能用’from module import \*’导入

\_\_xxx\_\_ 系统定义名字

\_\_xxx 类中的私有变量名

核心风格避免用下划线作为变量名的开始。

因为下划线对解释器有特殊的意义而且是内建标识符所使用的符号我们建议程序员避免用下划线作为变量名的开始。一般来讲变量名\_xxx被看作是“私有 的”在模块或类外不可以使用。当变量是私有的时候用\_xxx 来表示变量是很好的习惯。因为变量名\_\_xxx\_\_对Python 来说有特殊含义对于普通的变量应当避免这种命名风格。

“单下划线” 开始的成员变量叫做保护变量意思是只有类对象和子类对象自己能访问到这些变量  
“双下划线” 开始的是私有成员意思是只有类对象自己能访问连子类对象也不能访问到这个数据。

以单下划线开头\_foo的代表不能直接访问的类属性需通过类提供的接口进行访问不能用“from xxx import \*”而导入以双下划线开头的\_\_foo代表类的私有成员以双下划线开头和结尾的\_\_foo\_\_代表python里特殊方法专用的标识如 \_\_init\_\_代表类的构造函数。

现在我们来总结下所有的系统定义属性和方法 先来看下保留属性

>>> Class1.\_\_doc\_\_ # 类型帮助信息 'Class1 Doc.' >>> Class1.\_\_name\_\_ # 类型名称 'Class1' >>> Class1.\_\_module\_\_ # 类型所在模块 '\_\_main\_\_' >>> Class1.\_\_bases\_\_ # 类型所继承的基类 (<type 'object'>,) >>> Class1.\_\_dict\_\_ # 类型字典存储所有类型成员信息。 <dictproxy object at 0x00D3AD70> >>> Class1().\_\_class\_\_ # 类型 <class '\_\_main\_\_.Class1'> >>> Class1().\_\_module\_\_ # 实例类型所在模块 '\_\_main\_\_' >>> Class1().\_\_dict\_\_ # 对象字典存储所有实例成员信息。 {'i': 1234}

接下来是保留方法可以把保留方法分类

**类的基础方法**

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **目的** | **所编写代码** | **Python 实际调用** |
| **①** | 初始化一个实例 | x = MyClass() | [x.\_\_init\_\_()](http://docs.python.org/3.1/reference/datamodel.html#object.__init__) |
| **②** | 字符串的“官方”表现形式 | repr(x) | [x.\_\_repr\_\_()](http://docs.python.org/3.1/reference/datamodel.html#object.__repr__) |
| **③** | 字符串的“非正式”值 | [str(x)](http://docs.python.org/3.1/reference/datamodel.html#object.__str__) | x.\_\_str\_\_() |
| **④** | 字节数组的“非正式”值 | bytes(x) | x.\_\_bytes\_\_() |
| **⑤** | 格式化字符串的值 | format(x, format\_spec) | [x.\_\_format\_\_(format\_spec)](http://docs.python.org/3.1/reference/datamodel.html#object.__format__) |

1. 对 \_\_init\_\_() 方法的调用发生在实例被创建 *之后* 。如果要控制实际创建进程请使用 [\_\_new\_\_() 方法](http://www.4pang.com/2010/08/20/4%E8%83%96%E5%AD%A6python-%EF%BC%8D-%E7%89%B9%E6%AE%8A%E5%B1%9E%E6%80%A7%E5%92%8C%E6%96%B9%E6%B3%95%E7%9A%84%E6%80%BB%E7%BB%93.html#esoterica)。
2. 按照约定 \_\_repr\_\_() 方法所返回的字符串为合法的 Python 表达式。
3. 在调用 print(x) 的同时也调用了 \_\_str\_\_() 方法。
4. 由于 bytes 类型的引入而*从 Python 3 开始出现*。

**行为方式与迭代器类似的类**

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **目的** | **所编写代码** | **Python 实际调用** |
| **①** | 遍历某个序列 | iter(seq) | [seq.\_\_iter\_\_()](http://www.python.org/doc/3.1/reference/datamodel.html#object.__iter__) |
| **②** | 从迭代器中获取下一个值 | next(seq) | [seq.\_\_next\_\_()](http://www.python.org/doc/3.1/reference/datamodel.html#object.__next__) |
| **③** | 按逆序创建一个迭代器 | reversed(seq) | [seq.\_\_reversed\_\_()](http://www.python.org/doc/3.1/reference/datamodel.html#object.__reversed__) |

1. 无论何时创建迭代器都将调用 \_\_iter\_\_() 方法。这是用初始值对迭代器进行初始化的绝佳之处。
2. 无论何时从迭代器中获取下一个值都将调用 \_\_next\_\_() 方法。
3. \_\_reversed\_\_() 方法并不常用。它以一个现有序列为参数并将该序列中所有元素从尾到头以逆序排列生成一个新的迭代器。

**计算属性**

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **目的** | **所编写代码** | **Python 实际调用** |
| **①** | 获取一个计算属性无条件的 | x.my\_property | [x.\_\_getattribute\_\_('my\_property')](http://www.python.org/doc/3.1/reference/datamodel.html#object.__getattribute__) |
| **②** | 获取一个计算属性后备 | x.my\_property | [x.\_\_getattr\_\_('my\_property')](http://www.python.org/doc/3.1/reference/datamodel.html#object.__getattr__) |
| **③** | 设置某属性 | x.my\_property = value | [x.\_\_setattr\_\_('my\_property',value)](http://www.python.org/doc/3.1/reference/datamodel.html#object.__setattr__) |
| **④** | 删除某属性 | del x.my\_property | [x.\_\_delattr\_\_('my\_property')](http://www.python.org/doc/3.1/reference/datamodel.html#object.__delattr__) |
| **⑤** | 列出所有属性和方法 | dir(x) | [x.\_\_dir\_\_()](http://www.python.org/doc/3.1/reference/datamodel.html#object.__dir__) |

1. 如果某个类定义了 \_\_getattribute\_\_() 方法在 *每次引用属性或方法名称时* Python 都调用它特殊方法名称除外因为那样将会导致讨厌的无限循环。
2. 如果某个类定义了 \_\_getattr\_\_() 方法Python 将只在正常的位置查询属性时才会调用它。如果实例 x 定义了属性color x.color 将 *不会* 调用x.\_\_getattr\_\_('color')而只会返回x.color 已定义好的值。
3. 无论何时给属性赋值都会调用 \_\_setattr\_\_() 方法。
4. 无论何时删除一个属性都将调用 \_\_delattr\_\_() 方法。
5. 如果定义了 \_\_getattr\_\_() 或 \_\_getattribute\_\_() 方法 \_\_dir\_\_() 方法将非常有用。通常调用 dir(x) 将只显示正常的属性和方法。如果\_\_getattr()\_\_方法动态处理color 属性 dir(x) 将不会将 color 列为可用属性。可通过覆盖 \_\_dir\_\_() 方法允许将 color 列为可用属性对于想使用你的类但却不想深入其内部的人来说该方法非常有益。

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **目的** | **所编写代码** | **Python 实际调用** |
|  | 序列的长度 | len(seq) | [seq.\_\_len\_\_()](http://www.python.org/doc/3.1/reference/datamodel.html#object.__len__) |
|  | 了解某序列是否包含特定的值 | x in seq | [seq.\_\_contains\_\_(x)](http://www.python.org/doc/3.1/reference/datamodel.html#object.__contains__) |

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **目的** | **所编写代码** | **Python 实际调用** |
|  | 通过键来获取值 | x[key] | [x.\_\_getitem\_\_(key)](http://www.python.org/doc/3.1/reference/datamodel.html#object.__getitem__) |
|  | 通过键来设置值 | x[key] = value | [x.\_\_setitem\_\_(key,value)](http://www.python.org/doc/3.1/reference/datamodel.html#object.__setitem__) |
|  | 删除一个键值对 | del x[key] | [x.\_\_delitem\_\_(key)](http://www.python.org/doc/3.1/reference/datamodel.html#object.__delitem__) |
|  | 为缺失键提供默认值 | x[nonexistent\_key] | [x.\_\_missing\_\_(nonexistent\_key)](http://docs.python.org/3.1/library/collections.html#collections.defaultdict.__missing__) |

**可比较的类**

我将此内容从前一节中拿出来使其单独成节是因为“比较”操作并不局限于数字。许多数据类型都可以进行比较——字符串、列表甚至字典。如果要创建自己的类且对象之间的比较有意义可以使用下面的特殊方法来实现比较。

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **目的** | **所编写代码** | **Python 实际调用** |
|  | 相等 | x == y | [x.\_\_eq\_\_(y)](http://www.python.org/doc/3.1/reference/datamodel.html#object.__eq__) |
|  | 不相等 | x != y | [x.\_\_ne\_\_(y)](http://www.python.org/doc/3.1/reference/datamodel.html#object.__ne__) |
|  | 小于 | x < y | [x.\_\_lt\_\_(y)](http://www.python.org/doc/3.1/reference/datamodel.html#object.__lt__) |
|  | 小于或等于 | x <= y | [x.\_\_le\_\_(y)](http://www.python.org/doc/3.1/reference/datamodel.html#object.__le__) |
|  | 大于 | x > y | [x.\_\_gt\_\_(y)](http://www.python.org/doc/3.1/reference/datamodel.html#object.__gt__) |
|  | 大于或等于 | x >= y | [x.\_\_ge\_\_(y)](http://www.python.org/doc/3.1/reference/datamodel.html#object.__ge__) |
|  | 布尔上上下文环境中的真值 | if x: | [x.\_\_bool\_\_()](http://www.python.org/doc/3.1/reference/datamodel.html#object.__bool__) |

**可序列化的类**

Python 支持 [任意对象的序列化和反序列化](http://blog.163.com/jackylau_v/blog/serializing.html)。多数 Python 参考资料称该过程为 “pickling” 和 “unpickling”。该技术对与将状态保存为文件并在稍后恢复它非常有意义。所有的 [内置数据类型](http://blog.163.com/jackylau_v/blog/native-datatypes.html) 均已支持 pickling 。如果创建了自定义类且希望它能够 pickle阅读 [pickle 协议](http://docs.python.org/3.1/library/pickle.html) 了解下列特殊方法何时以及如何被调用。

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **目的** | **所编写代码** | **Python 实际调用** |
|  | 自定义对象的复制 | copy.copy(x) | [x.\_\_copy\_\_()](http://docs.python.org/3.1/library/copy.html) |
|  | 自定义对象的深度复制 | copy.deepcopy(x) | [x.\_\_deepcopy\_\_()](http://docs.python.org/3.1/library/copy.html) |
|  | 在 pickling 之前获取对象的状态 | pickle.dump(x, file) | [x.\_\_getstate\_\_()](http://docs.python.org/3.1/library/pickle.html#pickle-state) |
|  | 序列化某对象 | pickle.dump(x, file) | [x.\_\_reduce\_\_()](http://docs.python.org/3.1/library/pickle.html#pickling-class-instances) |
|  | 序列化某对象新 pickling 协议 | pickle.dump(x, file, protocol\_version) | [x.\_\_reduce\_ex\_\_(protocol\_version)](http://docs.python.org/3.1/library/pickle.html#pickling-class-instances) |
| **\*** | 控制 unpickling 过程中对象的创建方式 | x = pickle.load(file) | [x.\_\_getnewargs\_\_()](http://docs.python.org/3.1/library/pickle.html#pickling-class-instances) |
| **\*** | 在 unpickling 之后还原对象的状态 | x = pickle.load(file) | [x.\_\_setstate\_\_()](http://docs.python.org/3.1/library/pickle.html#pickle-state) |

\* 要重建序列化对象Python 需要创建一个和被序列化的对象看起来一样的新对象然后设置新对象的所有属性。\_\_getnewargs\_\_() 方法控制新对象的创建过程而 \_\_setstate\_\_() 方法控制属性值的还原方式。

**可在**with**语块中使用的类**

with 语块定义了 [运行时刻上下文环境](http://www.python.org/doc/3.1/library/stdtypes.html#typecontextmanager)在执行 with 语句时将“进入”该上下文环境而执行该语块中的最后一条语句将“退出”该上下文环境。

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **目的** | **所编写代码** | **Python 实际调用** |
|  | 在进入 with 语块时进行一些特别操作 | with x: | [x.\_\_enter\_\_()](http://www.python.org/doc/3.1/reference/datamodel.html#object.__enter__) |
|  | 在退出 with 语块时进行一些特别操作 | with x: | [x.\_\_exit\_\_()](http://www.python.org/doc/3.1/reference/datamodel.html#object.__exit__) |

以下是 [with file 习惯用法](http://blog.163.com/jackylau_v/blog/files.html#with) 的运作方式

# excerpt from io.py: def \_checkClosed(self, msg=None):     '''Internal: raise an ValueError if file is closed     '''     if self.closed:         raise ValueError('I/O operation on closed file.'                          if msg is None else msg)  def \_\_enter\_\_(self):     '''Context management protocol.  Returns self.'''     self.\_checkClosed()                                ①     return self                                        ②  def \_\_exit\_\_(self, \*args):     '''Context management protocol.  Calls close()'''     self.close()                                       ③

1. 该文件对象同时定义了一个 \_\_enter\_\_() 和一个 \_\_exit\_\_() 方法。该 \_\_enter\_\_() 方法检查文件是否处于打开状态如果没有 \_checkClosed()方法引发一个例外。
2. \_\_enter\_\_() 方法将始终返回 self —— 这是 with 语块将用于调用属性和方法的对象
3. 在 with 语块结束后文件对象将自动关闭。怎么做到的在 \_\_exit\_\_() 方法中调用了 self.close() .

?该 \_\_exit\_\_() 方法将总是被调用哪怕是在 with 语块中引发了例外。实际上如果引发了例外该例外信息将会被传递给 \_\_exit\_\_() 方法。查阅 [With 状态上下文环境管理器](http://www.python.org/doc/3.1/reference/datamodel.html#with-statement-context-managers) 了解更多细节。

**真正神奇的东西**

如果知道自己在干什么你几乎可以完全控制类是如何比较的、属性如何定义以及类的子类是何种类型。

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **目的** | **所编写代码** | **Python 实际调用** |
|  | 类构造器 | x = MyClass() | [x.\_\_new\_\_()](http://www.python.org/doc/3.1/reference/datamodel.html#object.__new__) |
| **\*** | 类析构器 | del x | [x.\_\_del\_\_()](http://www.python.org/doc/3.1/reference/datamodel.html#object.__del__) |
|  | 只定义特定集合的某些属性 |  | [x.\_\_slots\_\_()](http://www.python.org/doc/3.1/reference/datamodel.html#object.__slots__) |
|  | 自定义散列值 | hash(x) | [x.\_\_hash\_\_()](http://www.python.org/doc/3.1/reference/datamodel.html#object.__hash__) |
|  | 获取某个属性的值 | x.color | [type(x).\_\_dict\_\_['color'].\_\_get\_\_(x, type(x))](http://www.python.org/doc/3.1/reference/datamodel.html#object.__get__) |
|  | 设置某个属性的值 | x.color = 'PapayaWhip' | [type(x).\_\_dict\_\_['color'].\_\_set\_\_(x, 'PapayaWhip')](http://www.python.org/doc/3.1/reference/datamodel.html#object.__set__) |
|  | 删除某个属性 | del x.color | [type(x).\_\_dict\_\_['color'].\_\_del\_\_(x)](http://www.python.org/doc/3.1/reference/datamodel.html#object.__delete__) |
|  | 控制某个对象是否是该对象的实例 your class | isinstance(x, MyClass) | [MyClass.\_\_instancecheck\_\_(x)](http://www.python.org/dev/peps/pep-3119/#overloading-isinstance-and-issubclass) |
|  | 控制某个类是否是该类的子类 | issubclass(C, MyClass) | [MyClass.\_\_subclasscheck\_\_(C)](http://www.python.org/dev/peps/pep-3119/#overloading-isinstance-and-issubclass) |
|  | 控制某个类是否是该抽象基类的子类 | issubclass(C, MyABC) | [MyABC.\_\_subclasshook\_\_(C)](http://docs.python.org/3.1/library/abc.html#abc.ABCMeta.__subclasshook__) |

python中以双下划线的是一些系统定义得名称让python以更优雅得语法实行一些操作本质上还是一些函数和变量与其他函数和变量无二。  
比如x.\_\_add\_\_(y) 等价于 x+y  
有一些很常见有一些可能比较偏在这里罗列一下做个笔记备忘。  
x.\_\_contains\_\_(y) 等价于 y in x, 在list,str, dict,set等容器中有这个函数  
\_\_base\_\_, \_\_bases\_\_, \_\_mro\_\_, 关于类继承和函数查找路径的。  
class.\_\_subclasses\_\_(), 返回子类列表  
x.\_\_call\_\_(...) == x(...)  
x.\_\_cmp\_\_(y) == cmp(x,y)  
x.\_\_getattribute\_\_('name') == x.name == getattr(x, 'name'),  比\_\_getattr\_\_更早调用  
x.\_\_hash\_\_() == hash(x)  
x.\_\_sizeof\_\_(), x在内存中的字节数, x为class得话 就应该是x.\_\_basicsize\_\_  
x.\_\_delattr\_\_('name') == del x.name  
\_\_dictoffset\_\_ attribute tells you the offset to where you find the pointer to the \_\_dict\_\_ object in any instance object that has one. It is in bytes.  
\_\_flags\_\_, 返回一串数字用来判断该类型能否被序列化if it's a heap type), \_\_flags\_\_ & 512  
S.\_\_format\_\_, 有些类有用  
x.\_\_getitem\_\_(y) == x[y], 相应还有\_\_setitem\_\_, 某些不可修改类型如setstr没有\_\_setitem\_\_  
x.\_\_getslice\_\_(i, j) == x[i:j], 有个疑问x='123456789', x[::2],是咋实现得  
\_\_subclasscheck\_\_(), check if a class is subclass  
\_\_instancecheck\_\_(), check if an object is an instance  
\_\_itemsize\_\_, These fields allow calculating the size in bytes of instances of the type. 0是可变长度 非0则是固定长度  
x.\_\_mod\_\_(y) == x%y, x.\_\_rmod\_\_(y) == y%x  
x.\_\_module\_\_ , x所属模块  
x.\_\_mul\_\_(y) == x\*y,  x.\_\_rmul\_\_(y) == y\*x

\_\_reduce\_\_, \_\_reduce\_ex\_\_ , for pickle

\_\_slots\_\_ 使用之后类变成静态一样没有了\_\_dict\_\_, 实例也不可新添加属性

\_\_getattr\_\_ 在一般的查找属性查找不到之后会调用此函数

\_\_setattr\_\_ 取代一般的赋值操作如果有此函数会调用此函数 如想调用正常赋值途径用 object.\_\_setattr\_\_(self, name, value)

\_\_delattr\_\_ 同\_\_setattr\_\_, 在del obj.name有意义时会调用

## 下面将讲解几个常用的：

**\_\_init\_\_():  
\_\_ init\_\_方法在类的一个对象被建立时，马上运行。这个方法可以用来对你的对象做一些你希望的初始化。注意，这个名称的开始和结尾都是双下划线。**

**class Person:**

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

**self.name = name**

**def sayHi(self):**

**print ('Hello, my name is', self.name)**

**说明：\_\_ init\_\_ 方法定义为取一个参数name（以及普通的参数self）。在这个\_\_ init\_\_ 里，我们只是创建一个新的域，也称为name。注意它们是两个不同的变量，尽管它们有相同的名字。点号使我们能够区分它们。最重要的是，我们没有专门调用\_\_ init\_\_ 方法，只是在创建一个类的新实例的时候，把参数包括在圆括号内跟在类名后面，从而传递给\_\_ init\_\_ 方法。这是这种方法的重要之处。现在，我们能够在我们的方法中使用self.name域。这在sayHi方法中得到了验证。**

**\_\_new\_\_ ()  
　　\_\_ new\_\_ ()在\_\_ init\_\_()之前被调用，用于生成实例对象.利用这个方法和类属性的特性可以实现设计模式中的单例模式.单例模式是指创建唯一对象吗，单例模式设计的类只能实例化一个对象.**

**class Singleton(object):**

**\_\_instance = None # 定义实例**

**def \_\_init\_\_(self):**

**pass**

**def \_\_new\_\_(cls, \*args, \*\*kwd): # 在\_\_init\_\_之前调用**

**if Singleton.\_\_instance is None: # 生成唯一实例**

**Singleton.\_\_instance = object.\_\_new\_\_(cls, \*args, \*\*kwd)**

**return Singleton.\_\_instance**

**\_\_str\_\_ ()  
　　\_\_ str\_\_ ()用于表示对象代表的含义，返回一个字符串.实现了\_\_ str\_\_ ()方法后，可以直接使用print语句输出对象，也可以通过函数str()触发\_\_ str\_\_ ()的执行.这样就把对象和字符串关联起来，便于某些程序的实现，可以用这个字符串来表示某个类**

**class Cat:**

**"""定义了一个Cat类"""**

**#初始化对象**

**def \_\_init\_\_(self, new\_name, new\_age):**

**self.name = new\_name**

**self.age = new\_age**

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

**return "%s的年龄是:%d"%(self.name, self.age)**

**\_\_del\_\_()**

**\_\_ del\_\_称作析构方法  
　　析构方法，当对象在内存中被释放时，自动触发执行。  
　　　　注：此方法一般无须定义，因为在Python中，程序员在使用时无需关心内存的分配和释放，因为此工作都是交给Python解释器来执行，所以，析构函数的调用是由解释器在进行垃圾回收时自动触发执行的。在程序执行结束之后，执行此方法**

**class Foo:**

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

**print('run \_\_del\_\_')**

**\_\_getattribute\_\_:**

**class Itcast(object):**

**def \_\_init\_\_(self,subject1):**

**self.subject1 = subject1**

**self.subject2 = 'cpp'**

**#属性访问时拦截器，打log**

**def \_\_getattribute\_\_(self,obj):**

**if obj == 'subject1':**

**print('log subject1')**

**return 'redirect python'**

**else: #测试时注释掉这2行，将找不到subject2**

**return object.\_\_getattribute\_\_(self,obj)**

**def show(self):**

**print('this is Itcast')**

**s = Itcast("python")**

**print(s.subject1)**

**print(s.subject2)**

**运行结果:**

**log subject1**

**redirect python**

**cpp**

**\_\_getattribute\_\_的坑**

**class Person(object):**

**def \_\_getattribute\_\_(self,obj):**

**print("---test---")**

**if obj.startswith("a"):**

**return "hahha"**

**else:**

**return self.test**

**def test(self):**

**print("heihei")**

**t.Person()**

**t.a #返回hahha**

**t.b #会让程序死掉**

**#原因是：当t.b执行时，会调用Person类中定义的\_\_getattribute\_\_方法，但是在这个方法的执行过程中**

**#if条件不满足，所以 程序执行else里面的代码，即return self.test 问题就在这，因为return 需要把**

**#self.test的值返回，那么首先要获取self.test的值，因为self此时就是t这个对象，所以self.test就是**

**#t.test 此时要获取t这个对象的test属性，那么就会跳转到\_\_getattribute\_\_方法去执行，即此时产**

**#生了递归调用，由于这个递归过程中 没有判断什么时候推出，所以这个程序会永无休止的运行下去，又因为**

**#每次调用函数，就需要保存一些数据，那么随着调用的次数越来越多，最终内存吃光，所以程序 崩溃**

**#**

**# 注意：以后不要在\_\_getattribute\_\_方法中调用self.xxxx**