## defaultdict()

在**[Python](http://lib.csdn.net/base/python" \o "Python知识库" \t "http://blog.csdn.net/real_ray/article/details/_blank)**中如果访问字典中不存在的键，会引发KeyError异常（**[JavaScript](http://lib.csdn.net/base/javascript" \o "JavaScript知识库" \t "http://blog.csdn.net/real_ray/article/details/_blank)**中如果对象中不存在某个属性，则返回undefined）。但是有时候，字典中的每个键都存在默认值是非常方便的。

### 1.1 默认值可以很方便

众所周知，在Python中如果访问字典中不存在的键，会引发KeyError异常（JavaScript中如果对象中不存在某个属性，则返回undefined）。但是有时候，字典中的每个键都存在默认值是非常方便的。例如下面的例子：

strings = ('puppy', 'kitten', 'puppy', 'puppy',

'weasel', 'puppy', 'kitten', 'puppy')

counts = {}

for kw in strings:

counts[kw] += 1

该例子统计strings中某个单词出现的次数，并在counts字典中作记录。单词每出现一次，在counts相对应的键所存的值数字加1。但是事实上，运行这段代码会抛出KeyError异常，出现的时机是每个单词第一次统计的时候，因为python的dict中不存在默认值的说法，可以在Python命令行中验证：

>>> counts = dict()

>>> counts

{}

>>> counts['puppy'] += 1

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

KeyError: 'puppy'

### 1.2 使用判断语句

既然如此，首先可能想到的方法是在单词第一次统计的时候，在counts中相应的键存下默认值1。这需要在处理的时候添加一个判断语句：

strings = ('puppy', 'kitten', 'puppy', 'puppy',

'weasel', 'puppy', 'kitten', 'puppy')

counts = {}

for kw in strings:

if kw not in counts:

counts[kw] = 1

else:

counts[kw] += 1

# counts:

# {'puppy': 5, 'weasel': 1, 'kitten': 2}

### 1.3 使用dict.setdefault()方法设置默认值

也可以通过dict.setdefault()方法来设置默认值：

strings = ('puppy', 'kitten', 'puppy', 'puppy',

'weasel', 'puppy', 'kitten', 'puppy')

counts = {}

for kw in strings:

counts.setdefault(kw, 0)

counts[kw] += 1

dict.setdefault()方法接收两个参数，第一个参数是健的名称，第二个参数是默认值。假如字典中不存在给定的键，则返回参数中提供的默认值；反之，则返回字典中保存的值。利用dict.setdefault()方法的返回值可以重写for循环中的代码，使其更加简洁：

strings = ('puppy', 'kitten', 'puppy', 'puppy',

'weasel', 'puppy', 'kitten', 'puppy')

counts = {}

for kw in strings:

counts[kw] = counts.setdefault(kw, 0) + 1

### 1.4 使用collections.defaultdict类

以上的方法虽然在一定程度上解决了dict中不存在默认值的问题，但是这时候我们会想，有没有一种字典它本身提供了默认值的功能呢？答案是肯定的，那就是collections.defaultdict。

defaultdict类就好像是一个dict，但是它是使用一个类型来初始化的：

>>> from collections import defaultdict

>>> dd = defaultdict(list)

>>> dd

defaultdict(<type 'list'>, {})

defaultdict类的初始化函数接受一个类型作为参数，当所访问的键不存在的时候，可以实例化一个值作为默认值：

>>> dd['foo']

[]

>>> dd

defaultdict(<type 'list'>, {'foo': []})

>>> dd['bar'].append('quux')

>>> dd

defaultdict(<type 'list'>, {'foo': [], 'bar': ['quux']})

需要注意的是，这种形式的默认值只有在通过dict[key]或者dict.\_\_getitem\_\_(key)访问的时候才有效，这其中的原因在下文会介绍。

>>> from collections import defaultdict

>>> dd = defaultdict(list)

>>> 'something' in dd

False

>>> dd.pop('something')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

KeyError: 'pop(): dictionary is empty'

>>> dd.get('something')

>>> dd['something']

[]

defaultdict类除了接受类型名称作为初始化函数的参数之外，还可以使用任何不带参数的可调用函数，到时该函数的返回结果作为默认值，这样使得默认值的取值更加灵活。下面用一个例子来说明，如何用自定义的不带参数的函数zero()作为defaultdict类的初始化函数的参数：

>>> from collections import defaultdict

>>> def zero():

... return 0

...

>>> dd = defaultdict(zero)

>>> dd

defaultdict(<function zero at 0xb7ed2684>, {})

>>> dd['foo']

0

>>> dd

defaultdict(<function zero at 0xb7ed2684>, {'foo': 0})

利用collections.defaultdict来解决最初的单词统计问题，代码如下：

from collections import defaultdict

strings = ('puppy', 'kitten', 'puppy', 'puppy',

'weasel', 'puppy', 'kitten', 'puppy')

counts = defaultdict(lambda: 0) # 使用lambda来定义简单的函数

for s in strings:

counts[s] += 1

### 1.5 defaultdict类是如何实现的

通过上面的内容，想必大家已经了解了defaultdict类的用法，那么在defaultdict类中又是如何来实现默认值的功能呢？这其中的关键是使用了看\_\_missing\_\_()这个方法：

>>> from collections import defaultdict>>> print defaultdict.\_\_missing\_\_.\_\_doc\_\_

\_\_missing\_\_(key) # Called by \_\_getitem\_\_ for missing key; pseudo-code:

if self.default\_factory is None: raise KeyError(key)

self[key] = value = self.default\_factory()

return value

通过查看\_\_missing\_\_()方法的docstring，可以看出当使用\_\_getitem\_\_()方法访问一个不存在的键时(dict[key]这种形式实际上是\_\_getitem\_\_()方法的简化形式)，会调用\_\_missing\_\_()方法获取默认值，并将该键添加到字典中去。

关于\_\_missing\_\_()方法的具体介绍可以参考Python官方文档中的"[Mapping Types — dict](http://docs.python.org/library/stdtypes.html" \l "dict" \t "http://blog.csdn.net/real_ray/article/details/_blank)"一节。

文档中介绍，从2.5版本开始，如果派生自dict的子类定义了\_\_missing\_\_()方法，当访问不存在的键时，dict[key]会调用\_\_missing\_\_()方法取得默认值。

从中可以看出，虽然dict支持\_\_missing\_\_()方法，但是在dict本身是不存在这个方法的，而是需要在派生的子类中自行实现这个方法。可以简单的验证这一点：

>>> print dict.\_\_missing\_\_.\_\_doc\_\_Traceback (most recent call last):

File "<stdin>", line 1, in <module>AttributeError: type object 'dict' has no attribute '\_\_missing\_\_'

同时，我们可以进一步的做实验，定义一个子类Missing并实现\_\_missing\_\_()方法:

>>> class Missing(dict):... def \_\_missing\_\_(self, key):... return 'missing'...>>> d = Missing()>>> d{}>>> d['foo']'missing'>>> d{}

返回结果反映了\_\_missing\_\_()方法确实发挥了作用。在此基础上，我们稍许修改\_\_missing\_\_()方法,使得该子类同defautldict类一样为不存在的键设置一个默认值：

>>> class Defaulting(dict):... def \_\_missing\_\_(self, key):... self[key] = 'default'... return 'default'...>>> d = Defaulting()>>> d{}>>> d['foo']'default'>>> d{'foo': 'default'}

## eval()和exec()、compile()

假如我们一串字符串里面有Python代码，这个时候，普通情况是会把这串代码作为字符串来输出的，而不会执行这段代码。如果此时，我们想执行这串字符串里面的Python代码，使用exec语句即可解决。

假如我们一串字符串里面有Python的表达式，这个时候，普通情况是会把这串表达式作为字符串来输出的，而不会执行这个表达式。如果此时，我们想执行这串字符串里面的Python表达式，使用eval语句即可解决。

exec与eval语句的主要区别是，exec处理字符串里面的代码，没有返回值，而eval是处理字符串里面的表达式，有返回值。

### 2.1 eval

功能： 将字符串str当成有效的表达式来求值并返回计算结果。

语法： eval(source[, globals[, locals]]) -> value

参数：

source：一个Python表达式或函数compile()返回的代码对象

globals：可选。必须是dictionary

locals：可选。任意map对象

# -\*- coding:utf-8 -\*-

# 字符串转换成列表

a = *"[[1,2], [3,4], [5,6], [7,8], [9,0]]"*

print type(a) # <type 'str'>

b = eval(a)

print b # [[1, 2], [3, 4], [5, 6], [7, 8], [9, 0]]

print type(b) # <type 'list'>

# test golbals() and locals()

x = 1

y = 1

num1 = eval(*"x+y"*)

print num1 # 2

def **g**():

x = 2

y = 2

num2 = eval(*"x+y"*)

print num2 # 4

num\_gol = eval(*"x+y"*, globals())

print num\_gol # 2

num\_local = eval(*"x+y"*, locals())

print num\_local # 4

g()

*eval有安全性问题,比如用户恶意输入就会获得当前目录文件:*

print eval(*"\_\_import\_\_('os').system('dir')"*)

### 2.2 exec

exec(object[, globals[, locals]])

本函数是执行一段语句或函数。参数object是一个字符串的语句或者一个编译过的语句的对象名称。参数globals是全局命名空间，用来指定执行语句时可以访问的全局命名空间；参数locals是局部命名空间，用来指定执行语句时可以访问的局部作用域的命名空间。要注意**本函数不会返回任何值**，不管函数或语句有任何的返回值语句，比return或yield语句。

# -\*- coding:utf-8 -\*-

exec(*'if True: print 100'*)

exec(*'''*

*x = 200*

*if x > 100:*

*print(x + 200)*

*'''*)

例：

C:\E\file.py

print "Hello Worlw!"

print "My Test!"

>>> exec(open(r'C:\E\file.py').read()) # 读取文件中字符串，并执行文件内语句

Hello Worlw!

My Test!

>>> exec "for i in range(5): print i"

0

1

2

3

4

### 2.3 compile

描述：compile() 函数将一个字符串编译为字节代码。

语法：

compile(...)

compile(source, filename, mode[, flags[, dont\_inherit]]) -> code object

Compile the source string (a Python module, statement or expression)

into a code object that can be executed by the exec statement or eval().

The filename will be used for run-time error messages.

The mode must be 'exec' to compile a module, 'single' to compile a

single (interactive) statement, or 'eval' to compile an expression.

The flags argument, if present, controls which future statements influence

the compilation of the code.

The dont\_inherit argument, if non-zero, stops the compilation inheriting

the effects of any future statements in effect in the code calling

compile; if absent or zero these statements do influence the compilation,

in addition to any features explicitly specified.

参数：

* source -- 字符串或者AST（Abstract Syntax Trees）对象。。
* filename -- 代码文件名称，如果不是从文件读取代码则传递一些可辨认的值。
* mode -- 指定编译代码的种类。可以指定为 exec, eval, single。
* flags -- 变量作用域，局部命名空间，如果被提供，可以是任何映射对象。。
* flags和dont\_inherit是用来控制编译源码时的标志

返回值：返回表达式执行结果

>>> S = "for i in range(10): print i"

>>> C = compile(S, filename="", mode="exec")

>>> C

<code object <module> at 0000000002F39E30, file "", line 1>

>>> C.co\_code

'x\x19\x00e\x00\x00d\x00\x00\x83\x01\x00D]\x0b\x00Z\x01\x00e\x01\x00GHq\r\x00Wd\x01\x00S'

>>> exec(c)

0

1

2

3

4

## cmp函数

cmp(...)

cmp(x, y) -> integer

Return negative if x<y, zero if x==y, positive if x>y.

如果比较的元素是同类型的,则比较其值,返回结果。

如果两个元素不是同一种类型,则检查它们是否是数字。

如果是数字,执行必要的数字强制类型转换,然后比较。

如果有一方的元素是数字,则另一方的元素"大"(数字是"最小的")

否则,通过类型名字的字母顺序进行比较。

如果有一个列表首先到达末尾,则另一个长一点的列表"大"。

如果我们用尽了两个列表的元素而且所 有元素都是相等的,那么结果就是个平局,就是说返回一个 0。

## 列表sort方法与sorted方法

### 4.1 list.sort

sort(...)

L.sort(cmp=None, key=None, reverse=False) -- stable sort \*IN PLACE\*;

cmp(x, y) -> -1, 0, 1

* cmp(仅支持Python2) : 比较函数，可以自定义，如cmp = lambda x,y:y-x; cmp = cmp(x,y)
* key : 排序关键字,值为一个函数，此函数只有一个参数且返回一个值，比如，有一个User对象的list，希望通过对象的 user\_id 属性进行排序，可以提供一个以 User 实例作为输入并输出对应 user\_id 值的函数给 key（即key = lambda u:u.user\_id 或 key=attrgetter(‘user\_id’)）
* reverse : 默认为False，为True的话就是反序排序

>>> L=[[1, 2, 3], [4, 5, 6], [3, 3, 3]]

>>> L.sort(cmp=lambda x, y: y-x, key=lambda x: x[1], reverse=True)

>>>

>>> L

[[1, 2, 3], [3, 3, 3], [4, 5, 6]]

### 4.2 sorted函数

Python的一个内置函数，使用方法与list.sort()大体一致，不同在于两个地方：

* sorted(L)返回一个排序后的L，不改变原始的L；L.sort()是对原始的L进行操作，调用后原始的L会改变，没有返回值。【所以a = a.sort()是错的啦！a = sorted(a)才对！
* sorted()适用于任何可迭代容器，list.sort()仅支持list（本身就是list的一个方法）

基于以上两点，sorted使用频率比list.sort()更高些，所以Python中更高级的排序技巧便通过sorted()来演示。

sorted(...)

sorted(iterable, cmp=None, key=None, reverse=False) --> new sorted list

## 内置函数

### 4.1 locals()

locals()函数返回当前函数（方法）的局部命名空间。

def func(a = 1):

b = 2 print(locals())

return a+b func() # {'a': 1, 'b': 2}

可以看出，locals返回的是个dict

>>> def func():

X = 2

locals()['Y'] = 100

print locals()

>>> func()

{'Y': 100, 'X': 2}

### 4.2 globals()

返回当前module的命名空间.

>>>X = 1

>>> def func():

X = 2

print globals()

>>> func()

{'\_\_builtins\_\_': <module '\_\_builtin\_\_' (built-in)>, '\_\_package\_\_': None, 'func': <function func at 0x00000000025F0AC8>, 'X': 1, '\_\_name\_\_': '\_\_main\_\_', '\_\_doc\_\_': None}

': None}

>>> globals()['Y'] = 3

>>> globals()

{'\_\_builtins\_\_': <module '\_\_builtin\_\_' (built-in)>, '\_\_package\_\_': None, 'func': <function func at 0x00000000025F0AC8>, 'X': 1, 'Y': 3, '\_\_name\_\_': '\_\_main\_\_', '\_\_doc\_\_': None}

## 数学操作函数

### math.floor()、math.trunc、round函数区别

**math.floor(x)**:返回x的下舍整数，小于等于x的最大整数。

>>> math.floor(-2.4)

-3.0

>>> math.floor(2.4)

2.0

**math.ceil(x):**取大于等于x的最小整数。

>>> math.ceil(-2.4)

-2.0

>>> math.ceil(2.4)

3.0

**math.trunc(x)**:该函数不对指定小数前或后的部分做相应舍入选择处理，而统统截去。

>>> math.trunc(-2.6)

-2

>>> math.trunc(3.4)

3

**round(x):**返回一个数的四舍五入值。

>>> round(2.4)

2.0

>>> round(-2.5)

-3.0

## type函数

>>> help(type)

Help on class type in module \_\_builtin\_\_:

class type(object)

| type(object) -> the object's type # 返回对象的类型

| type(name, bases, dict) -> a new type # 返回一个新的类型（类）

|

| Methods defined here:

|

| \_\_call\_\_(...)

| x.\_\_call\_\_(...) <==> x(...)

|

| \_\_delattr\_\_(...)

| x.\_\_delattr\_\_('name') <==> del x.name

|

| \_\_eq\_\_(...)

| x.\_\_eq\_\_(y) <==> x==y

|

| \_\_ge\_\_(...)

| x.\_\_ge\_\_(y) <==> x>=y

|

| \_\_getattribute\_\_(...)

| x.\_\_getattribute\_\_('name') <==> x.name

|

| \_\_gt\_\_(...)

| x.\_\_gt\_\_(y) <==> x>y

|

| \_\_hash\_\_(...)

| x.\_\_hash\_\_() <==> hash(x)

|

| \_\_init\_\_(...)

| x.\_\_init\_\_(...) initializes x; see help(type(x)) for signature

|

| \_\_instancecheck\_\_(...)

| \_\_instancecheck\_\_() -> bool

| check if an object is an instance

|

| \_\_le\_\_(...)

| x.\_\_le\_\_(y) <==> x<=y

|

| \_\_lt\_\_(...)

| x.\_\_lt\_\_(y) <==> x<y

|

| \_\_ne\_\_(...)

| x.\_\_ne\_\_(y) <==> x!=y

|

| \_\_repr\_\_(...)

| x.\_\_repr\_\_() <==> repr(x)

|

| \_\_setattr\_\_(...)

| x.\_\_setattr\_\_('name', value) <==> x.name = value

|

| \_\_subclasscheck\_\_(...)

| \_\_subclasscheck\_\_() -> bool

| check if a class is a subclass

|

| \_\_subclasses\_\_(...)

| \_\_subclasses\_\_() -> list of immediate subclasses

|

| mro(...)

| mro() -> list

| return a type's method resolution order

|

| ----------------------------------------------------------------------

| Data descriptors defined here:

|

| \_\_abstractmethods\_\_

|

| \_\_base\_\_

|

| \_\_bases\_\_

|

| \_\_basicsize\_\_

|

| \_\_dict\_\_

|

| \_\_dictoffset\_\_

|

| \_\_flags\_\_

|

| \_\_itemsize\_\_

|

| \_\_mro\_\_

|

| \_\_weakrefoffset\_\_

|

| ----------------------------------------------------------------------

| Data and other attributes defined here:

|

| \_\_new\_\_ = <built-in method \_\_new\_\_ of type object>

| T.\_\_new\_\_(S, ...) -> a new object with type S, a subtype of T

* name -- 类的名称。
* bases -- 基类的元组。
* dict -- 字典，类内定义的命名空间变量。

一个参数返回对象类型, 三个参数，返回新的类型对象。

>>> type(int)

<type 'type'> # 返回值是type类

>>> type(1) # 返回一个int类，int类是type类的一个实例

<type 'int'>

>>> type(int) == type(1)

False

>>> I = type(1)(10) # I是int类的一个实例，即10

>>> I

10

>>>F oo = type('Foo', (), {'bar': True}) # 创建一个名为Foo的类对象

>>> Foo1.bar

True

>>> Foo1.\_\_name\_\_

等价于：

>>> class Foo(object):

bar = True

继承Foo类，并添加一个方法：

>>> def echo(self):

print self.bar

>>> FooChild = type('FooChild', (Foo,), {'echo': echo}) # 创建了一个类FooChild,是Foo类的子类，并添加了echo方法。

>>> FooChild.bar

True

>>> X = FooChild()

>>> X.echo()

True

## hasattr、setattr、getattr函数

### setattr函数

setattr(...)

setattr(object, name, value)

Set a named attribute on an object; setattr(x, 'y', v) is equivalent to

``x.y = v''.

参数：

* object -- 对象。
* name -- 字符串，对象属性。
* value -- 属性值。

>>> class Car:

pass

>>> setattr(Car, 'name', 'benz') # 设置类对象的属性

>>> Car.name

'benz'

>>> c = Car()

>>> setattr(c, 'num', 10010) # 设置实例对象的属性

>>> c.num

10010

### getattr函数

>>> help(getattr)

Help on built-in function getattr in module \_\_builtin\_\_:

getattr(...)

getattr(object, name[, default]) -> value

Get a named attribute from an object; getattr(x, 'y') is equivalent to x.y.

When a default argument is given, it is returned when the attribute doesn't

exist; without it, an exception is raised in that case.

参数：

* object -- 对象。
* name -- 字符串，对象属性。
* default-- 默认值。

获取对象object的属性或者方法，如果存在打印出来，如果不存在，打印出默认值，默认值可选。  
需要注意的是，如果是返回的对象的方法，返回的是方法的内存地址。

>>> getattr(Car, 'name')

'benz'

>>> getattr(Car, 'model') # 属性不存在则抛出异常

Traceback (most recent call last):

File "<pyshell#25>", line 1, in <module>

getattr(Car, 'model')

AttributeError: class Car has no attribute 'model'

>>> getattr(Car, 'model', 'X') # 如果有默认值的获取默认值

'X'

>>> def run(self):

print self.name

>>> setattr(Car, 'run', run) # 设置方法

>>> c.run()

benz

>>> getattr(c, 'run')

<bound method Car.run of <\_\_main\_\_.Car instance at 0x0000000002FB1988>>

>>> getattr(Car, 'run')

<unbound method Car.run>

### hasattr

>>> help(hasattr)

Help on built-in function hasattr in module \_\_builtin\_\_:

hasattr(...)

hasattr(object, name) -> bool

Return whether the object has an attribute with the given name.

(This is done by calling getattr(object, name) and catching exceptions.)

判断一个对象里面是否有name属性或者name方法，返回BOOL值，有name特性返回True， 否则返回False。

>>> hasattr(Car, 'run')

True

>>> hasattr(Car, 'running')

False