```
import numpy as np # 在使用本文档前记得预先导入所有需要的包
```

More Language Features

本章提供了更多的Python语法特性，需要细细品读以体会Pythonic Coding的细节。

如何处理错误？

```
def var(y):
    n = len(y)
    # assert 后面加的判断语句是我们希望成立的条件，如果不成立就输出语句尾部的字符串作为错误提示
    assert n > 1, 'Sample size must be greater than one.'
    return np.sum((y - y.mean())**2) / float(n-1) # 注意此处的类型是np.ndarray
```

```
var(y = np.array([1, 2])) # 可以捕捉到assert语句中提到的错误，运行终止
```

```
0.5
```

在程序运行的同时处理错误

在Python中常见的错误有NameError, TypeError, IndexError, ZeroDivisionError等

```
def f(x):
    # 使用try-except模块来捕捉出问题的地方
    # 如果try语句中正确执行，那么就不会执行except中的语句；
    # 如果try中的语句出现错误，那么执行except中的语句
    # 和if-else的区别在于这个语句不会打断程序的执行，只是在出现错误的时候输出想要输出的结果
    try:
        return 1.0 / x
    except ZeroDivisionError: # 这个地方的错误名是固定的
        print('Error: division by zero. Returned None')
    except TypeError: # 可以添加多种错误情况
        print('Error: Unsupported operation. Returned None')
    return None
```

```
type(f(0))
type(f('a'))
```

```
Error: division by zero. Returned None
Error: Unsupported operation. Returned None
```

```
NoneType
```

装饰器（Decorator）

装饰器存在的意义：本质上来说装饰器就是一种函数闭包，即一种从函数映射到新函数的特殊函数；其意义在于我们能够很方便地对已存在的函数进行修改，即装饰，故曰装饰器。

```
# 首先我们得到两个用于演示的示例函数
def f(x):
    return np.log(x) # x > 1

def g(x):
    return np.sqrt(42 * x) # x > 0
```

现在我们希望对上述两个函数做出修改如下：在执行计算前先判断是否在函数定义域内，否则返回相应错误信息。

一种简单的方式当然是对上面的函数进行改动，但我们希望有一种更优雅的方式：即使用装饰器对函数进行修改。

```
def check_nonneg(f):

    def safe_f(y): # 这就是我们将要返回的新函数
        print('message')
        assert y > 0, "Argument must be nonnegative!" # 也可以通过try-except去catch
errors
        return f(y) # 它的命名空间可以被新的函数记住

    return safe_f
```

```
safe_f = check_nonneg(f) # 每次执行该语句都会在check_nonneg(f) 的函数体中定义新的
safe_f()并返回
```

```
safe_f(1)
```

```
message



0.0
```

到现在为止都还很好理解，现在让我们来看一些奇怪的事情。我们知道，在Python中一切都是对象。我们在check_nonneg中调用了对象f，生成了新的对象。但是在我们直接使用新对象（即装饰过后的函数）的时候，却可以从容地使用原来函数的功能，这是为什么呢？

Python支持一个叫做函数闭包的功能。在以函数作为返回值的时候，可以记住该函数的命名空间。

```
f = check_nonneg(f) # 演示，多次调用可以记住每一层的命名空间
f(1)
```

```
message
message
message
message
message



0.0
```

更令人惊讶的是，Python的开发者考虑地是如此的周全，以至于他们把函数装饰这样一个常用的功能直接集成到了Python的syntax之中。注意，以下表达虽然和上面的有所不同，但这只是我们书写方式的问题；而在interpreter的眼中，它们是完完全全等价的。

```
@check_nonneg
def h(x):
    return abs(np.log(x))
```

```
h(0)
```

```
message



---------------------------------------------------------------------------

AssertionError                            Traceback (most recent call last)

<ipython-input-16-201738d38ff9> in <module>()
----> 1 h(0)


<ipython-input-7-315bf2a5c0ce> in safe_f(y)
      3     def safe_f(y): # 这就是我们将要返回的新函数
      4         print('message')
----> 5         assert y > 0, "Argument must be nonnegative!" # 也可以通过try-
except去catch errors
      6         return f(y) # 它的命名空间可以被新的函数记住
      7


AssertionError: Argument must be nonnegative!
```

描述（Descriptor）

描述符只是Python中的一种对象，它被创造来让我们能够更方便地管理变量。首先让我们看看官方的文档是怎么说的。

In general, a descriptor is an object attribute with “binding behavior”, one whose attribute access has been overridden by methods in the descriptor protocol. **Those methods are `__get__()`, `__set__()`, and `__delete__()`. If any of those methods are defined for an object, it is said to be a descriptor**.

Data and non-data descriptors differ in how overrides are calculated with respect to entries in an instance’s dictionary. If an instance’s dictionary has an entry with the same name as a data descriptor, the data descriptor takes precedence. If an instance’s dictionary has an entry with the same name as a non-data descriptor, the dictionary entry takes precedence.

Invoking descriptors

一个描述可以直接通过它的方法名被调用。例如 `d.__get__(obj)`

```
class Car:

    def __init__(self, miles=1000):
        self._miles = miles
        self._kms = miles * 1.61

    def set_miles(self, value):
```

```
        self._miles = value
        self._kms = value * 1.61

    def set_kms(self, value):
        self._kms = value
        self._miles = value / 1.61

    def get_miles(self):
        return self._miles

    def get_kms(self):
        return self._kms

    miles = property(get_miles, set_miles)
    kms = property(get_kms, set_kms)
```

```
car = Car()
car.miles
```

```
1000
```

```
car.miles = 6000 # 这是我们希望达成的效果
car.kms
```

```
9660.0
```

```
car.__dict__['_kms']
```

```
9660.0
```

```
type(car).__dict__ # 相当于获取类Class的namespace
```

```
mappingproxy({'__module__': '__main__',
              '__init__': <function __main__.Car.__init__(self, miles=1000)>,
              'set_miles': <function __main__.Car.set_miles(self, value)>,
```

```
              'set_kms': <function __main__.Car.set_kms(self, value)>,
              'get_miles': <function __main__.Car.get_miles(self)>,
              'get_kms': <function __main__.Car.get_kms(self)>,
              'miles': <property at 0x2ab32031c78>,
              'kms': <property at 0x2ab32031d18>,
              '__dict__': <attribute '__dict__' of 'Car' objects>,
              '__weakref__': <attribute '__weakref__' of 'Car' objects>,
              '__doc__': None})
```

```
class Test(object):
    cls_val = 1 # 这玩意儿叫做类属性
    def __init__(self):
        self.ins_val = 10
```

```
t = Test()
```

```
print(t.__dict__)
print(Test.__dict__)
```

```
{'ins_val': 10}
{'__module__': '__main__', 'cls_val': 1, '__init__': <function Test.__init__ at
0x000002AB3205C158>, '__dict__': <attribute '__dict__' of 'Test' objects>,
'__weakref__': <attribute '__weakref__' of 'Test' objects>, '__doc__': None}
```

```
Test.cls_val
```

```
1
```

```
print(t.__getattribute__('ins_val'))
print(Test.__getattribute__('cls_val')) # 这是一个类，而不是一个实例
```

```
10
```

```
--------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-26-dd4edb7e1e14> in <module>()
      1 print(t.__getattribute__('ins_val'))
----> 2 print(Test.__getattribute__('cls_val')) # 这是一个类，而不是一个实例


TypeError: expected 1 arguments, got 0
```

```
t.__getattribute__('cls_val') # 会在父类中寻找
```

```
1
```

```
class Desc(object):

    def __get__(self, instance, owner):
        print("__get__...")
        print("self : \t\t", self)
        print("instance : \t", instance)
        print("owner : \t", owner)
        print('='*40, "\n")

    def __set__(self, instance, value):
        print('__set__...')
        print("self : \t\t", self)
        print("instance : \t", instance)
        print("value : \t", value)
        print('='*40, "\n")


class TestDesc(object):
    x = Desc() # 注意这里x是类的属性而不是所建实例的属性

#以下为测试代码
t = TestDesc()
t.x
```

```
__get__...
self :          <__main__.Desc object at 0x000002AB31FB5F98>
instance :      <__main__.TestDesc object at 0x000002AB32067B00>
owner :         <class '__main__.TestDesc'>
========================================
```

我们可以看出，在实例化 TestDesc 之后，调用对象并访问其属性 t，会自动调用类 Desc 的 __get__ 方法。从输出信息可以看出：

- self：其实就是Desc的实例化对象，即t.x
- instance：TestDesc的实例化对象，即t
- owner：最大的容器，即TestDesc这个类

但是问题又出现了，为什么在调用t.x的时候，直接执行了 __get__ 方法呢？

根据常规顺序，在访问t.x的时候，应该先在实例t的属性中搜索，没找到；然后继续在TestDesc的属性中寻找，找到了！

但此时解释器发现x这玩意居然是一个描述符，就会把 TestDesc.x 转化为 TestDesc.__dict__['x'].__get__(None, TestDesc) 来访问。用人话来说**也就是我们在访问一个描述符实例的时候，会自动变成访问它的 __get__ 函数**。

```

```

```
TestDesc.__dict__['x'].__get__(None, TestDesc)
```

```
__get__...
self :          <__main__.Desc object at 0x000002AB31FB5F98>
instance :      None
owner :         <class '__main__.TestDesc'>
========================================
```

装饰器之property

Python 总共有三个内置装饰器：

- staticmethod
- classmethod
- property

通常，在我们访问属性和进行属性赋值的时候，都要和类和实例的 __dict__ 打交道。但如果想要规范属性的访问，有两种方式：**数据描述符**、 **property() 函数**。但是描述符比较复杂，新手们不妨试试 property() 。

```
# 首先我们看看原始的储存数据的方法

class Stu():

    def __init__(self, score):
        self._score = score
```

```
s = Stu(90)
```

```
print(s.__dict__)
s._score
```

```
{'_score': 90}
```

```
90
```

```
# 假设我们希望在每次对数据进行赋值的时候进行某种操作：同时计算另一个值或者对输入的数据进行判断，要怎么办呢？

class Stu():

    def __init__(self):
        self._s = 100
        self._ss = self._s/10

    def gets(self):
        return self._s

    def sets(self, score):
        self._s = score

    def dels(self):
        del self

s = Stu()
```

```
s.sets(90)
print(s._s)
print(s._ss)
```

```
90
10.0
```

```
# 不妨想想，其实我们只要让函数在每次赋值的时候有一个“同步”操作就可以了，是不是和数据描述符的功能很像？

class Stu():

    def __init__(self):
        self._s = 100
        self._ss = self._s/10

    def gets(self):
        return self._s

    def sets(self, score):
        self._s = score
        self._ss = score/10

    def dels(self):
        del self

    def getss(self):
        return self._ss

    def setss(self, score):
        self._ss = score
        self._s = score*10

    def delss(self):
        del self

    # 到这里为止一切都没有改变，我们加上下面这句
    y = property(gets, sets, dels, "I'm a property")
    z = property(getss, setss, delss, "I'm a property")
```

```
# 现在我们可以像正常的实例属性一样访问y，但实际上y的内容存在另一个实例属性中，故对数据的操作得到了控制
s = Stu()
s.y = 90
s.z
```

```
9.0
```

```
# 我们可以看到property函数本身是一种装饰符，于是我们可以用相同的简化方式

class Stuu():
```

```
    def __init__(self):
        self._s = 100
        self._ss = self._s/10

    @property
    def x(self):
        return self._s

    # @property.setter
```

现在我们已经对 property 的使用有了一个大概的印象。但这个东西到底是什么呢？我们可以来看看他的 python 等效实现代码：

```
class Property(object):
    "Emulate PyProperty_Type() in Objects/descrobject.c"

    def __init__(self, fget=None, fset=None, fdel=None, doc=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
        if doc is None and fget is not None:
            doc = fget.__doc__
        self.__doc__ = doc

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        if self.fget is None:
            raise AttributeError("unreadable attribute")
        return self.fget(obj)

    def __set__(self, obj, value):
        if self.fset is None:
            raise AttributeError("can't set attribute")
        self.fset(obj, value)

    def __delete__(self, obj):·
        if self.fdel is None:
            raise AttributeError("can't delete attribute")
        self.fdel(obj)

    def getter(self, fget):
        return type(self)(fget, self.fset, self.fdel, self.__doc__)

    def setter(self, fset):
        return type(self)(self.fget, fset, self.fdel, self.__doc__)

    def deleter(self, fdel):
        return type(self)(self.fget, self.fset, fdel, self.__doc__)
```

```
  File "<ipython-input-38-afbd2855d318>", line 24
    def __delete__(self, obj):·
                               ^
SyntaxError: invalid character in identifier
```

明白了吧！property 本质上是一个数据描述符（定义见上文）。

产生器（Generator）

产生器本身就是一种迭代器（可以使用 next 方法）。我们有两种方法实现产生器：

- 产生器表达式
- 产生器函数

产生器表达式（generator expression）

```
singular = ('dog', 'cat', 'bird')
plural = (string + 's' for string in singular) # 注意，如果此处使用中括号 [] ，那么最终将生成list
type(plural)
```

```
generator
```

产生器函数（generator function）

利用表达式来生成产生器，其使用方法更像一种定式，不够灵活。采用函数来生成产生器有更大的灵活性。

```
def g(x):
    while x < 100:
        yield x
        x += 1

gen = g(0)
type(gen)
```

```
generator
```

明明知道可以用迭代器完成的任务，都可以用列表加循环来完成，那为什么还要用**迭代器/产生器**来完成呢？

这是由于迭代器只描述了一种产生数据的法则，在程序运行中不断产生数据，并且用完即丢，这将极大程度上减少对内存空间的使用。

递归

递归本身很好理解，即函数调用了自己本身。如果我们把函数想象成同层面一系列操作的集合，那么递归调用就会一直增加层数。相当于在内存中这开了一个栈（先进先出的数据结构，更多内容请参考《数据结构》一书），在运行的过程中不断加深，等在某个地方触底之后再回到上一层继续执行操作。

```
# 先通过一个简单的例子来说明递归在 Python 中是如何实现的

def x(t):
    if t == 0:
        return 1
    else:
        return 2 * x(t-1)

x(2)
```

```
4
```

我们可以看到这是一个递归，在写递归函数的时候，最重要的是必须写明**边界条件**，也就是递归调用在什么时候结束。如果没有停止条件，那么程序会不断调用自己，栈不断往下挖，这将陷入死循环。

练习题

Exercise 1

```
# 利用递归算法实现 Fibonacci 数列

def Fibo(n):
    x0 = 0
    x1 = 1
    if n >= 2: return Fibo(n-1) + Fibo(n-2)
    elif n == 1: return x1
    else: return x0
```

个人习惯是先写好动态方程，然后再补全停止条件；因为很难一开始就写全所有的停止条件。

```
Fibo(9)
```

```
34
```

Exercise 2

Complete the following code, and test it using this csv file, which we assume that you've put in your current working directory

```
def column_iterator(target_file, column_number):
    """A generator function for CSV files.
    When called with a file name target_file (string) and column number
    column_number (integer), the generator function returns a generator
    that steps through the elements of column column_number in file
    target_file.
    """
    f = open(target_file, 'r') # 只读文件
    for line in f:
        yield line.split('\n')[column_number - 1] # 如果把 \n 换成别的不会出现的字符，输出会增加换行
    f.close()



dates = column_iterator('us_cities.csv', 1)
for date in dates:
    print(date)
```

```
new york: 8244910
los angeles: 3819702
chicago: 2707120
houston: 2145146
philadelphia: 1536471
phoenix: 1469471
san antonio: 1359758
san diego: 1326179
dallas: 1223229
```

Exercise 3

```
def iterator(file):
    f = open(file)
    for line in f:
        try: yield int(line.split('\n')[0])
        except ValueError:
            print('Not integer!')
    f.close()
```

```
it = iterator('numbers.txt')
sum(x for x in it)
```

```
Not integer!
Not integer!




39
```