目录

[1.数据类型与变量 2](#_Toc17604)

[1.1类型判断 2](#_Toc6915)

[1.2类型转换 2](#_Toc14865)

[1.3 整除与取余 2](#_Toc16817)

[1.4 set数据类型 2](#_Toc24038)

[1.5 允许多个赋值在同一行a,b=b,a+b 2](#_Toc3006)

[1.6 list拷贝与C++不同 3](#_Toc26562)

[1.7数据类型之间的转换(list与str之间转换较难) 3](#_Toc7231)

[2.字符串 4](#_Toc31130)

[2.1字符串删除 4](#_Toc13255)

[2.2 转义字符用\或者字符串前面加r 4](#_Toc1655)

[2.3 对不可变变量的理解 4](#_Toc25326)

[2.4 格式化输出别用%控制，python2.6开始引进format（）功能强大好用 5](#_Toc1891)

[2.5 判断字符串是否是字母或数字 5](#_Toc1574)

[3函数 5](#_Toc14444)

[3.1默认参数有个巨大的坑 5](#_Toc32026)

[3.2可变参数 6](#_Toc9983)

[3.3 迭代 7](#_Toc25135)

[3.4 map()和reduce()简化了代码 8](#_Toc25586)

[4模块 8](#_Toc27358)

[4.1 模块中的公有变量，私有变量，特殊变量 8](#_Toc32493)

[5类 8](#_Toc8186)

[5.1可以自由地给一个实例变量绑定属性 8](#_Toc24085)

[5.2访问限制 9](#_Toc21802)

[5.3@property 10](#_Toc23597)

[6正则表达式 10](#_Toc16660)

[6.1对用户输入的字符串进行合法性检查 10](#_Toc10365)

[6.2切割字符串（很方便） 11](#_Toc25621)

[7时间 11](#_Toc32583)

[7.1时间的获取 11](#_Toc6516)

[附录1 12](#_Toc18066)

[1.1在根目录下搜索某个文件，返回其绝对路径 12](#_Toc20340)

[1.2 显示某个对象所拥有的全部属性和方法 12](#_Toc14990)

# 1.数据类型与变量

## 1.1类型判断

使用内建的isinstance函数可以判断一个变量是不是字符串：

>>>x='abc'

>>>y=123

>>>isinstance(x,str)

True

## 1.2类型转换

返回字符的ASCII值

>>> ord('a')

97

>>> chr(97)

'a'

int()除了可以将字符串转化为整数，int()函数还提供额外的 base 参数，默认值为 10。如果传入 base 参数，

就可以做 N 进制的转换：

>>> int('12345', base=8)

5349

>>> int('12345', 16)

74565

## 1.3 整除与取余

>>> 10//3

3

>>> 10%3

1

## 1.4 set数据类型

即数学上的集合，集合中的元素不能重复，且没有顺序，可以做交，并运算

>>> s1=set([1,2,3])

>>> s2=set([2,3,4])

>>> s1&s2

{2, 3}

>>> s1|s2

{1, 2, 3, 4}

## 1.5 允许多个赋值在同一行a,b=b,a+b

>>> a,b=0,1

>>> a

0

>>> b

1

>>> a,b=b,a+b

>>> a

1

>>> b

1

结果不是a=1,b=2的原因是等式两边的两次赋值是同时进行的

## 1.6 list拷贝与C++不同

a=[1,2,3]

b=a

上式其实是引用复制，b和a均是指向[1,2,3]的引用

要想拷贝一份，用:b=a.copy()

或者用列表生成式：b=[i for i in a]

或者最简单：b=a[:]

补充深拷贝与浅拷贝问题：

>>> import copy

>>> a = [1,2,3,4,5]

>>> b = ["A","B",a]

>>> #浅拷贝

>>> c = b[:]

>>> c

['A', 'B', [1, 2, 3, 4, 5]]

>>> a[0] = 11

>>> c

['A', 'B', [11, 2, 3, 4, 5]]

>>> #此时a变化c跟着变化

>>> #深拷贝

>>> c = copy.deepcopy(b)

>>> c

['A', 'B', [11, 2, 3, 4, 5]]

>>> a[0] = 111

>>> a

[111, 2, 3, 4, 5]

>>> c

['A', 'B', [11, 2, 3, 4, 5]]

>>> #此时c中数据不受a影响

## 1.7数据类型之间的转换(list与str之间转换较难)

l=[1,2,3]

strings='123'

t=(1,2,3)

s=set([1,2,3])

print('strings->list:',[int(i) for i in strings])

from functools import reduce

print('list->strings:',str(reduce(lambda a,b:a\*10+b,l)))

print('set->list:',list(s))

print('list->set:',set(l))

print('tuple->list:',list(t))

print('list->tuple:',tuple(l))

print('str->int:',int(strings),type(int(strings)))

print('str->float:',float(strings),type(float(strings)))

>>>

strings->list: [1, 2, 3]

list->strings: 123

set->list: [1, 2, 3]

list->set: {1, 2, 3}

tuple->list: [1, 2, 3]

list->tuple: (1, 2, 3)

str->int: 123 <class 'int'>

str->float: 123.0 <class 'float'>

## 1.8 多维列表

创建初始值为0的2x3x4规格的三维列表：

a=[[[0 for i in range(4)] for j in range(3)] for k in range(2)]

# 2.字符串

## 2.1字符串删除

字符串的话,你可以把他当作列表处理：

str='hello world'

如果是想去掉第一个字母'o'，并且知道是第5个字符,index=4

1.使用分片new\_str=str[:4]+str[5:];

2.循环读取new\_str= ''.join([str[i] for i in range(len(str)) if i!= 4]) ;

3.字符替换new\_str = str.replace('o','',1) #后面的1代表替换第一个匹配'o'的字符 .

## 2.2 转义字符用\或者字符串前面加r

>>> s2='Hello,\'Adam\''

>>> s3=r'Hello,"Bart"'

>>> print(s2)

Hello,'Adam'

>>> print(s3)

Hello,"Bart"

## 2.3 对不可变变量的理解

而对于不可变对象，比如 str，对 str 进行操作呢：

>>> a = 'abc'

>>> a.replace('a', 'A')

'Abc'

>>> a

'abc'

虽然字符串有个 replace()方法，也确实变出了'Abc'，但变量 a 最后仍

是'abc'，应该怎么理解呢？

我们先把代码改成下面这样：

>>> a = 'abc'

>>> b = a.replace('a', 'A')

>>> b

'Abc'

>>> a

'abc'

要始终牢记的是，a是变量，而'abc'才是字符串对象！有些时候，我们

经常说，对象a的内容是'abc'，但其实是指，a本身是一个变量，它指

向的对象的内容才是'abc'：

当我们调用 a.replace('a', 'A')时，实际上调用方法 replace 是作用在

字符串对象'abc'上的，而这个方法虽然名字叫replace，但却没有改变

字符串'abc'的内容。相反，replace方法创建了一个新字符串'Abc'并返

回，如果我们用变量 b 指向该新字符串，就容易理解了，变量a仍指向

原有的字符串'abc'，但变量b却指向新字符串'Abc'了：

所以，对于不变对象来说，调用对象自身的任意方法，也不会改变该对

象自身的内容。相反，这些方法会创建新的对象并返回，这样，就保证

了不可变对象本身永远是不可变的。

## 2.4 格式化输出别用%控制，python2.6开始引进format（）功能强大好用

## 2.5 判断字符串是否是字母或数字

x=’sh’

x.isalpha()

x=’344’

x.isdigit()

# 3函数

## 3.1默认参数有个巨大的坑

默认参数很有用，但使用不当，也会掉坑里。默认参数有个最大的坑，

演示如下：

先定义一个函数，传入一个 list，添加一个 END 再返回：

def add\_end(L=[]):

L.append('END')

return L

当你使用默认参数调用时，一开始结果也是对的：

>>> add\_end()

['END']

但是，再次调用 add\_end()时，结果就不对了：

>>> add\_end()

['END', 'END']

>>> add\_end()

['END', 'END', 'END']

很多初学者很疑惑，默认参数是[]，但是函数似乎每次都“记住了”上次

添加了'END'后的 list。

原因解释如下：

Python 函数在定义的时候，默认参数 L 的值就被计算出来了，即[]，因

为默认参数 L 也是一个变量，它指向对象[]，每次调用该函数，如果改

变了 L 的内容，则下次调用时，默认参数的内容就变了，不再是函数定

义时的[]了。

所以，定义默认参数要牢记一点：默认参数必须指向不变对象！

要修改上面的例子，def add\_end(L=None):

if L is None:

L = []

L.append('END')

return L

现在，无论调用多少次，都不会有问题我们可以用 None 这个不变对象来实现。

## 3.2可变参数

我们把函数的参数改为可变参数：

def calc(\*numbers):

sum = 0

for n in numbers:

sum = sum + n \* n

return sum

定义可变参数和定义一个 list 或 tuple 参数相比，仅仅在参数前面加了一

个\*号。在函数内部，参数 numbers 接收到的是一个 tuple，因此，函数

代码完全不变。但是，调用该函数时，可以传入任意个参数，包括 0 个

参数：

>>> calc(1, 2)

5

>>> calc()

0

如果已经有一个 list 或者 tuple，要调用一个可变参数怎么办？可以这样

做：

>>> nums = [1, 2, 3]

>>> calc(nums[0], nums[1], nums[2])

14

这种写法当然是可行的，问题是太繁琐，所以 Python 允许你在 list 或 tuple

前面加一个\*号，把 list 或 tuple 的元素变成可变参数传进去：

>>> nums = [1, 2, 3]

>>> calc(\*nums)

14

## 3.3 迭代

list 这种数据类型虽然有下标，但很多其他数据类型是没有下标的，但

是，只要是可迭代对象，无论有无下标，都可以迭代，比如 dict 就可以

迭代：

>>> d = {'a': 1, 'b': 2, 'c': 3}

>>> for key in d:

print(key)

...

a

c

b

因为 dict 的存储不是按照 list 的方式顺序排列，所以，迭代出的结果顺

序很可能不一样。

默认情况下， dict 迭代的是 key。如果要迭代 value，可以用 for value in

d.values()，如果要同时迭代 key 和 value，可以用 for k, v in d.items()。

由于字符串也是可迭代对象，因此，也可以作用于 for 循环：

>>> for ch in 'ABC':

print(ch)

...

A

B

C

最后一个小问题，如果要对 list 实现类似 Java 那样的下标循环怎么办？

Python 内置的 enumerate 函数可以把一个 list 变成索引-元素对，这样就

可以在 for 循环中同时迭代索引和元素本身：

>>> for i, value in enumerate(['A', 'B', 'C']):

... print(i, value)

...

0 A

1 B

2 C

上面的 for 循环里，同时引用了两个变量，在 Python 里是很常见的，比

如下面的代码：

>>> for x, y in [(1, 1), (2, 4), (3, 9)]:

print(x, y)

...  
1 1  
2 4  
3 9

## 3.4 map()和reduce()简化了代码

我们先看 map。 map()函数接收两个参数，一个是函数，一个是 Iterable，  
map 将传入的函数依次作用到序列的每个元素，并把结果作为新的  
Iterator 返回。

>>> def f(x):

... return x \* x

...

>>> r = map(f, [1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> list(r)

[1, 4, 9, 16, 25, 36, 49, 64, 81]

map()传入的第一个参数是 f，即函数对象本身。由于结果 r 是一个

Iterator， Iterator 是惰性序列，因此通过 list()函数让它把整个序列都

计算出来并返回一个 list。

再看 reduce 的用法。 reduce 把一个函数作用在一个序列[x1, x2, x3, ...]

上，这个函数必须接收两个参数， reduce 把结果继续和序列的下一个元

素做累积计算，其效果就是：

from functools import reduce

reduce(f, [x1, x2, x3, x4]) = f(f(f(x1, x2), x3), x4)

# 4模块

## 4.1 模块中的公有变量，私有变量，特殊变量

正常的函数和变量名是公开的（ public），可以被直接引用，比如： abc，

x123， PI 等；

类似\_\_xxx\_\_这样的变量是特殊变量，可以被直接引用，但是有特殊用途，

比如上面的\_\_author\_\_， \_\_name\_\_就是特殊变量， hello 模块定义的文档

注释也可以用特殊变量\_\_doc\_\_访问，我们自己的变量一般不要用这种变

量名；类似\_xxx 和\_\_xxx 这样的函数或变量就是非公开的（ private），不应该被直接引用，比如\_abc， \_\_abc 等；

之所以我们说， private 函数和变量“不应该”被直接引用，而不是“不能”

被直接引用，是因为 Python 并没有一种方法可以完全限制访问 private

函数或变量，但是，从编程习惯上不应该引用 private 函数或变量。

# 5类

## 5.1可以自由地给一个实例变量绑定属性

由于类可以起到模板的作用，因此，可以在创建实例的时候，把一些我

们认为必须绑定的属性强制填写进去。通过定义一个特殊的\_\_init\_\_方

法，在创建实例的时候，就把 name， score 等属性绑上去：

class Student(object):

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

self.name = name

self.score = score

注意到\_\_init\_\_方法的第一个参数永远是 self，表示创建的实例本身，

因此，在\_\_init\_\_方法内部，就可以把各种属性绑定到 self，因为 self

就指向创建的实例本身。（这与C++定义类中的数据成员很不同）

和静态语言不同， Python 允许对实例变量绑定任何数据，也就是说，对

于两个实例变量，虽然它们都是同一个类的不同实例，但拥有的变量名

称都可能不同：

>>> bart = Student('Bart Simpson', 59)

>>> lisa = Student('Lisa Simpson', 87)

>>> bart.age = 8

>>> bart.age

8

>>> lisa.age

Traceback (most recent call last):

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

AttributeError: 'Student' object has no attribute 'age'

## 5.2访问限制

如果要让内部属性不被外部访问，可以把属性的名称前加上两个下划线

\_\_，在 Python 中，实例的变量名如果以\_\_开头，就变成了一个私有变量

（ private），只有内部可以访问，外部不能访问，所以，我们把 Student

类改一改：

class Student(object):

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

self.\_\_name = name

self.\_\_score = score

def print\_score(self):

print('%s: %s' % (self.\_\_name, self.\_\_score))

改完后，对于外部代码来说，没什么变动，但是已经无法从外部访问实

例变量.\_\_name 和实例变量.\_\_score 了：

>>> bart = Student('Bart Simpson', 98)

>>> bart.\_\_name

Traceback (most recent call last):

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

AttributeError: 'Student' object has no attribute '\_\_name'

这样就确保了外部代码不能随意修改对象内部的状态，这样通过访问限

制的保护，代码更加健壮。

需要注意的是，在 Python 中，变量名类似\_\_xxx\_\_的，也就是以双下划

线开头，并且以双下划线结尾的，是特殊变量，特殊变量是可以直接访

问的，不是 private 变量。

有些时候，你会看到以一个下划线开头的实例变量名，比如\_name，这样的实例变量外部是可以访问的，但是，按照约定俗成的规定，当你看到这样的变量时，意思就是， “虽然我可以被访问，但是，请把我视为私有变量，不要随意访问”。

## 5.3@property

作用就是将类中的方法变成属性。

class Student(object):

@property

def score(self):

return self.\_score

@score.setter

def score(self, value):

if not isinstance(value, int):

raise ValueError('score must be an integer!')

if value < 0 or value > 100:

raise ValueError('score must between 0 ~ 100!')

self.\_score = value

@property 的实现比较复杂，我们先考察如何使用。把一个 getter 方法变

成属性，只需要加上@property 就可以了，此时， @property 本身又创建

了另一个装饰器@score.setter，负责把一个 setter 方法变成属性赋值，

于是，我们就拥有一个可控的属性操作：

>>> s = Student()

>>> s.score = 60 # OK，实际转化为 s.set\_score(60)

>>> s.score # OK，实际转化为 s.get\_score()

60

>>> s.score = 9999

Traceback (most recent call last):

...

ValueError: score must between 0 ~ 100!

总结：其实这段代码实现的功能和C++中的操作符重载达到的效果类似，其目的都是为了界面更加人性化。

@property 广泛应用在类的定义中，可以让调用者写出简短的代码，同时

保证对参数进行必要的检查，这样，程序运行时就减少了出错的可能性。

# 6正则表达式

## 6.1对用户输入的字符串进行合法性检查

先看看如何判断正则表达式是否匹配：

>>> import re

>>> re.match(r'^\d{3}\-\d{3,8}$', '010-12345')

<\_sre.SRE\_Match object; span=(0, 9), match='010-12345'>

>>> re.match(r'^\d{3}\-\d{3,8}$', '010 12345')

match()方法判断是否匹配，如果匹配成功，返回一个 Match 对象，否则

返回 None。常见的判断方法就是：

test = '用户输入的字符串'

if re.match(r'正则表达式', test):

print('ok')

else:

print('failed')

## 6.2切割字符串（很方便）

用正则表达式切分字符串比用固定的字符更灵活，请看正常的切分代

码：

>>> 'a b c'.split(' ')

['a', 'b', '', '', 'c']

嗯，无法识别连续的空格，用正则表达式试试：

>>> re.split(r'\s+', 'a b c')

['a', 'b', 'c']

无论多少个空格都可以正常分割。加入,试试：

>>> re.split(r'[\s\,]+', 'a,b, c d')

['a', 'b', 'c', 'd']

再加入;试试：

>>> re.split(r'[\s\,\;]+', 'a,b;; c d')

['a', 'b', 'c', 'd']

如果用户输入了一组标签，下次记得用正则表达式来把不规范的输入转

化成正确的数组。

# 7时间

## 7.1时间的获取

方法一

import time

print(time.time())

print(time.asctime())

print(time.strftime('%Y-%m-%d %H:%M:%S',time.localtime()))

>>>

1516067675.6436322

Tue Jan 16 09:54:35 2018

2018-01-16 09:54:35

方法二：

import datetime

print(datetime.date.today())

print(datetime.date.today().strftime('%d/%m/%Y'))

print(datetime.date(1941, 1, 5))

>>>

2018-01-16

16/01/2018

1941-01-05

# 8 python编程风格

## 8.1 模块布局

# (1) 起始行(Unix)

# (2) 模块文档Edit By Vheavens

# (3) 模块导入

# (4) 变量定义

# (5) 类定义

# (6) 函数定义

# (7) 主程序

推荐代码风格：主程序调用 main()函数

主程序代码通常都和你前面看到的代码相似，检查 \_\_name\_\_ 变量的值然后再执行相应的调用（参阅下一页的核心笔记）。主程序中的代码通常包括变量赋值， 类定义和函数定义，随后检查\_\_name\_\_来决定是否调用另一个函数（ 通常调用 main()函数）来完成该模块的功能。主程序通常都是做这些事。（我们上面的例子中使用 test()而不是 main()是为了避免你在读到核心笔记前感到迷惑。） 不管用什么名字， 我们想强调一点那就是：这儿是放置测试代码的好地方。我们在 3.4.2 小节中曾经说过，大部分的 Python 模块都是用于导入调用的，直接运行模块应该调用该模块的回归测试代码。

很多项目都是一个主程序，由它导入所有需要的模块。所以请记住，绝大部的模块创建的目的是为了被别人调用而不是作为独立执行的脚本。 我们也很可能创建一个 Python 库风格的模块，这种模块的创建目的就是为了被其他模块调用。总之，只有一个模块，也就是包含主程  
序的模块会被直接执行。

**核心笔记：\_\_name\_\_ 指示模块应如何被加载**

由于主程序代码无论模块是被导入还是被直接执行都会运行， 我们必须知道模块如何决定运行方向。一个应用程序可能需要导入另一个应用程序的一个模块，以便重用一些有用的代码（否则就只能用拷贝粘贴那种非面向对象的愚蠢手段）。 这种情况下，你只想访问那些位于其它应用程序中的代码，而不是想运行那个应用程序。因此一个问题出现了，“Python 是否有一种方法能在运行时检测该模块是被导入还是被直接执行呢？” 答案就是......(鼓声雷动).....没错！ \_\_name\_\_ 系统变量就是正确答案。

# 9 深度学习常用的库

## 9.1 collection

### 9.1.1 为多个种类计数

from collections import Counter

cnt = Counter()

for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']:

cnt[word] += 1

print cnt

输出：

Counter({'blue': 3, 'red': 2, 'green': 1})

# 附录1

## 1.1在根目录下搜索某个文件，返回其绝对路径

import os

import logging

print(os.listdir('.'))

os.chdir('F:/')

print(os.getcwd())

print([x for x in os.listdir('.') if os.path.isdir(x)])

def search\_file(fname,directory):

routes=[]

for root,dirs,files in os.walk(directory):

for f in files:

if(fname==f):

routes.append(root)

return routes

print(search\_file('trees.py','F:'))

## 1.2 显示某个对象所拥有的全部属性和方法

x=’23fdsf4’

dir(x)