### 变量和数据类型

##### 布尔类型

and 与运算

or 或运算

not 非运算

0 ，空字符串‘’ ，None 为False

其他数值、非空字符都是True

|  |
| --- |
| a=’t’  b=’e’  #若a为True，与运算结果取决于b,则结果将返回b的值；若a为False，与运算结果一定为False，短路计算，不考虑b了，所以返回a的值  print(a and b) # 输出e  print(a or b) # 输出t  a=’’  print a and b #输出 e |

##### print()

print 可以打印多个变量，用逗号分开，一个逗号显示一个空格

print(something,end=’’,sep=’’)

end参数：每行末尾追加的字符，默认是\n换行符。

sep参数：同时打印多个值，中间用什么断开，默认是一个空格

##### 数字

数据类型是不允许改变的,这就意味着如果改变数字数据类型得值，将重新分配内存空间。

int(x) 将x转化为整数

float(x) 将x转化为浮点数

/ 除，python3中结果总会返回一个浮点数，即便他是个整数

// 会返回一个舍弃了小数部分的整数

\*\* 幂运算

### List和Tuple类型

list为有序集合。类似于php数组，支持重复元素

list=[‘a’,’b’]

##### list中添加新元素

list.append(‘ele’) 在list末尾添加ele元素

list.insert(index,’ele’) 把新元素ele添加到list中索引时index的位置。

insert为前插，即插入到原来index那个元素的前面，所以list.insert(-1,’ele’),会插入到倒数第二个位置。

##### list中删除元素

del list[2] 删除第三个元素

list.pop() 删除list最后一个元素，并返回这个元素

list.pop(index) 删除index索引位置的元素，并返回这个元素

list.remove(obj) 移除列表中第一个obj匹配项

|  |
| --- |
| L = ['apple',100,0.01,['pear','A','b'],'C']  L.remove('apple')  print(L) |

##### 列表支持+ \*

+：合并两个list 返回一个新的list

|  |
| --- |
| list1=[1,2,3]  list2=[100,200,300]  list3=list1+list2  print(list3)  [1, 2, 3, 100, 200, 300] |

\*：复制列表

|  |
| --- |
| list4=['a','b','c']  list5=list4\*3  print(list5)  ['a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c'] |

##### 列表长度

len(list)

##### x in list

3 in [1,2,3] 返回True 元素是否位于列表中

##### 列表中的最大值，最小值

max(list)

min(list)

max(list, key, default)

求迭代器的最大值, max会for i in … 遍历一遍这个迭代器,然后将迭代器的每一个返回值当做参数传给key=func中的func（一般是lambda表达式）然后将执行结果传给key，然后以key为标准进行大小判断。

|  |
| --- |
| d1 = {'name': 'egon', 'price': 100}  d2 = {'name': 'rdw', 'price': 666}  d3 = {'name': 'zat', 'price': 1}  L1 = [d1, d2, d3]  a = max(L1, key=lambda x: x['name'])  print(a)  b = max(L1, key=lambda x: x['price'])  print(b)  结果：  {'name': 'zat', 'price': 1}  {'name': 'rdw', 'price': 666}  例2：  L = [100,30,’40’]  max(L,key=int)  当没有最大值时，必须设置默认值default=xxx |

对于字符串的比较，是从左边第一个字符开始比，按其ASKII码大小比较，若第一个字符相同，再去比第二个字符。

##### 列表扩展 改变原来的list

list.extend(extendList)

|  |
| --- |
| aList=[1,2,3]  b=[4,5,6]  aList.extend(b)  print(aList) 直接改变了原list |

##### 返回某个元素的索引

list.index(obj)

|  |
| --- |
| list = ['apple',100,0.01,['pear','A','b'],'C']  print(list.index('apple')) |

##### 统计某个元素在列表中有多少个

|  |
| --- |
| aList=['to','do','or','not','to','do','to']  print(aList.count('to')) |

##### reverse()

list.reverse() 将列表元素翻转，无返回值，改变原数组

list.sort() 将列表升序排列

list.sort(reverse=True) 降续排列

##### 创建tuple

**判断元组的关键是逗号 ，并非小括号**

|  |
| --- |
| a= 1,2,3 #不加括号也是个元祖  print(type(a))  tuple |

tuple也是有序列表，中文翻译为‘元祖’，一旦创建不能修改。

t = ('Adam', 'Lisa', 'Bart')

tuple也可以用t[2]这样访问，就是不能修改。

创建单元素tuple时，元素后要多加个“，”

t=(1,) 跟运算符（）以示区分

tuple的不可修改，是指向不变，但是指向的内容可以更改。

|  |
| --- |
| t = ('a', 'b', ['A', 'B'])  L[0] = 'X'  L[1] = 'Y'  print t  ('a', 'b', ['X', 'Y'])  #这里第三个元素还是指向list，只是list里的元素变了，但指向并没改变 |

元组中的元素值是不允许修改的，但我们可以对元组进行连接组合,使用+

元祖中的元素值时不能被删除的，但可以用del删除整个元组

### 条件判断和循环

关于缩进，python中，相同缩进的代码视为代码块，缩进使用四个空格，不要使用Tab，更不要混合使用。

##### if

|  |
| --- |
| if a>b:  print(…) |

##### if else

|  |
| --- |
| if a>b:  …  elif a>c:  ….  else:  …. |

##### for循环

|  |
| --- |
| L = ['a','b','c','d']  for name in L: #name为每个元素的变量  print(name)  for …:  语句  else: #以上循环终止时执行，但若循环时被break终止的，不执行else  语句 |

##### while

|  |
| --- |
| while 判断条件：  语句  while 判断条件：  语句  else： #当判断条件为False时，执行else语句  语句 |

##### 三元操作符

var = x if x>y else y

### dict和set类型

##### dict 字典

带索引的集合，形式：

|  |
| --- |
| d = {  'Adam': 95,  'Lisa': 85,  'Bart': 59  } |

访问时可以用d[‘Adan’] ,但是若这个key不存在，直接报错。

所以可以使用：

if ‘Adam’in d:

…

d.get(‘Adam’) //存在这个key，返回value，不存在，返回None

d.get(‘Adam’,’Not Found’) //第二个参数为默认值，找不到时返回

d. setdefault(key,default) 同get类似，如果key存在于字典当中，返回对应的值，

若不存在，则插入key并设置默认值default,并返回default值

##### dict特点

1. 查找速度快
2. key不能重复
3. key-value序对是没有顺序的
4. 作为key的元素必须不可变，list不能作为key

##### dict是无序的，若想让他按添加的顺序进行有序排列：

|  |
| --- |
| from collections import OrderedDict    orderDict=OrderedDict()  orderDict['a']=1  orderDict['b']=2  orderDict['c']=3  print(orderDict)  >>>OrderedDict([('a', 1), ('b', 2), ('c', 3)]) |

##### 字典推导式

|  |
| --- |
| students\_score={'jack':80,'james':91,'leo':100,'sam':60}  print({key:val for key,val in students\_score.items() if val>90})  类似于列表生成式:第一部分是字典的中每个元素的形式，第二部分是迭代，第三部分是if语句过滤 |

##### 字典更新

|  |
| --- |
| dict = {'Name': 'Runoob', 'Age': 7,'Sex':'male'}  dict2 = {'Height':177 ,'Sex': 'female'}  dict.update(dict2)  print(dict)  {'Height': 177, 'Name': 'Runoob', 'Age': 7, 'Sex': 'female'} |

将dict2中的元素更新到dict中，若dict2中的key在dict中存在，则更新dict该key的value，若不存在，则添加到dict中。

d3 = dict(d1, \*\*d2) 合并d1和d2

##### 字典删除

|  |
| --- |
| dict = {1,2,[3,6,9]}}}  dict2 = dict  dict.clear()  print(dict, dict2)  输出[] []  clear能删除内存上的内容。所以通过赋值产生的变量一样被清空了  而dict ={}，仅仅是断开了list跟内存之间的链接，并没有删除内存上的内容 |

dict.pop(key) 根据key值删除一个键值对，返回被删除的这个value

dict.popitem() 随机删除一个键值对，并返回这个键值对

##### fromkeys() 创建一个新字典

dict.fromkeys(seq[, value]))

seq作为新dict的键，value作为所有键的值

seq可以是string，list，tuple，但value只能是string或number

|  |
| --- |
| x={}  x = x.fromkeys(['a','b'],3)  print(x)  {'b': 3, 'a': 3}  但不能x = x.fromkeys(['a','b'],[3,4]) |

##### set

无序不重复的集合

s = {'A', 'C', 'B'}

或s = set([‘A’,’B’,’C’]) //set()里传入一个list

{'A', 'C', 'B'}

注意：创建一个空集合，必须用set()而不能用{}，因为{}用来创建一个空字典

‘A’ in s 返回True或False ,判断一个元素是否在set中

特点：

1. 元素必须是不可变对象
2. 无序的

集合不支持索引

所以不能用set\_arr[0]这样取集合里的元素

向set添加元素：

s.add(‘D’) //若添加的元素已存在在set中，不会报错，但也不会再添加进去一次了

从set删除元素：

s.remove(‘D’) //若要删除的元素不存在，会报错

所以，add（）可以直接添加，remove()前需要先判断

##### set交集和并集

s1 = {1, 2, 3}

s2 = {2, 3, 4}

print(s1 & s2) 交集 输出：{2, 3}

print(s1|s2) 并集 输出：{1, 2, 3, 4}

### 函数

##### 定义函数

def 函数名(形参):

‘函数文档’

…

return …

函数文档是一个字符串，不会输出，可以用\_\_doc\_\_访问到

##### 可变参数

def fn(\*args):

print args

Python解释器会把传入的一组参数组装成一个tuple传递给可变参数，因此，在函数内部，直接把变量 args 看成一个 tuple 就好了。

def fn(\*args,\*\*kwargs)

print kwargs

kwargs 关键字参数 key word arguments

kwargs 是个dict集合

##### 函数参数

* + 1. 位置参数

|  |
| --- |
| def fn(a,b):  print(a,b)  fn(1,2) |

函数调用的时候，所有参数在传递的时候按照位置来传递，此处把1赋给a,把2赋给b

* + 1. 默认参数

|  |
| --- |
| def fn(a,b=2):  print(a,b)  fn(1) |

b有默认参数，调用的时候可以不传

* + 1. 关键字参数

通过变量名进行匹配.

|  |
| --- |
| def fn(a,b=2,c=5):  print(a,b,c)  fn(4,c=9,b=3) 4没有用关键字，必须放在最前面，若想让a也放后面，下面这样：  fn(c=9,b=3,a=4) |

* + 1. 任意多个位置参数的函数

|  |
| --- |
| def fn(\*a):  print(a)  fn('2','a','b')  输出：('2', 'a', 'b')  \*a表示可以跟任意数量的位置参数  python会自动把\*操作符后面的形参变成tuple传给函数.，调用时把所有位置参数变成tuple传进a |

* + 1. 任意多个关键字参数的函数

|  |
| --- |
| def fn(\*\*b):  print(b)  fn(x='2',y='a',z='b')  输出：{'x': '2', 'y': 'a', 'z': 'b'}  python把\*\*后面的变量参数，按照字典来处理,传递给函数. |

##### return 多个值

|  |
| --- |
| def fn():  return 'xxxxxxxxxx',True  a,b=fn()  print(a,b) #xxxxxxxxxx True |

函数可以有多个返回值

原理：函数返回了一个元祖tuple，然后把结果赋给多个变量

小技巧：

有时候我们希望丢弃一些返回值，可以用\_作为变量（\_是个几乎不用的变量名）

|  |
| --- |
| def fn():  return 'xxxxxxxxxx',True,False  \_,a,b=fn()  print(a,b) |

##### 动态默认参数

|  |
| --- |
| import datetime,time  def log(message,when=datetime.datetime.now()):  print('%s:%s' %(when,message))  log('hello')  time.sleep(1)  log('hello')  结果：  2018-01-06 14:57:47.891158:hello  2018-01-06 14:57:47.891158:hello |

上面本想打印出隔一秒钟的时间，却输出相同的时间。因为datetime.datetime.now()只在函数定义的时候执行了一次，参数的默认值会在每一个模块加载进来的时候求出，一旦这段模块加载进来了，参数的默认值就很固定了，程序不会再出执行datetime.datetime.now()

所以若想动态实现默认值，习惯把默认值设为None

|  |
| --- |
| import datetime,time  def log(message,when=None):  when=datetime.datetime.now()  print('%s:%s' %(when,message))  log('hello')  time.sleep(1)  log('hello') |

### 切片

##### 对list切片

前包含后不包含

L[0:3] 从索引0开始，到索引3，但不包括3

L[:3] 若第一个索引是0，也可以省略

L[:] 表示从头到尾

L[::3]表示每三个元素取出一个，即隔两个取一个（三个参数）

##### 对字符串切片

字符串也可以看成list进行切片

'ABCDEFG'[:3] 返回结果还是字符串

### 迭代

##### 集合

集合是指包含一组元素的数据结构，我们已经介绍的包括：

1. 有序集合：list，tuple，str和unicode；

2. 无序集合：set

3. 无序集合并且具有 key-value 对：dict

他们都可以用for in 来迭代

##### for value in L

迭代取出来的永远是元素本身，不是索引

要想取索引：使用enumerate(L) L作为参数

|  |
| --- |
| L = ['Adam', 'Lisa', 'Bart', 'Paul']  for t in enumerate(L):  print t  输出：  (0, 'Adam')  (1, 'Lisa')  (2, 'Bart')  (3, 'Paul') |

enumerate将list有序集合每个元素都变成了tuple （index,value）

可以进一步简化：

|  |
| --- |
| L = ['Adam', 'Lisa', 'Bart', 'Paul']  for index,value in enumerate(L):  print index,'-',value |

index代表索引，value代表元素

##### zip（）

将两个list合并为一个list

|  |
| --- |
| zip([10, 20, 30], ['A', 'B', 'C'])  结果：[(10, 'A'), (20, 'B'), (30, 'C')] |

##### dict迭代

|  |
| --- |
| d = { 'Adam': 95, 'Lisa': 85, 'Bart': 59, 'Paul': 74 }  for key in d:  print key  结果：Lisa  Paul  Adam  Bart |

dict 迭代里的i，是key，并不是value

要想获取value，使用dict的values()方法，实际上是将所有的value值组合成一个list；

要获取key，使用dict.keys()

items() 作为字典的一个方法，字典在它前面

将dict转化成包含tuple的list

|  |
| --- |
| d = { 'Adam': 95, 'Lisa': 85, 'Bart': 59 }  print d.items()  [('Lisa', 85), ('Adam', 95), ('Bart', 59)]  然后通过  for key,value in d.items():  …  循环 |

##### 列表生成式

range(1,101,2) 生成一个list集合，从1到100（不含101），且间隔是1

第一个参数是从几开始，

第二个参数是到几结束（不包含），

第三个参数是每几个数取一个出来

列表生成式格式：[x\*x for x in L]

前面是列表中的元素，后面跟for循环

后面还可以跟if判断语句

[x\*x for x in L if x%2==0] 只有满足if条件才将x返回到列表

还可以循环嵌套，只要在后面继续跟for in 即可

[x+y+z for x in L for y in M for z in N]

等同于

|  |
| --- |
| A=[]  for x in L:  for y in M:  for z in N:  A.append(x+y+z) |

##### 生成器

通过列表生成器可以直接创建一个列表，但是受到内存限制，列表容量是有限的，若要创建一个含有100万个元素的列表，不仅占用很大的存储空间，如果我们仅仅需要访问前面几个元素，后面绝大多数的占用空间都白白浪费了。

所以，如果列表元素可以按照某种算法推算出来，那我们就可以在循环的过程中不断推算出后续的元素，这样就不必创建完整的list，从而节省大量的空间。

在Python中，这种一边循环一边计算的机制，称为生成器：generator

第一种创建方法：

把列表生成器的 [] 改成（）

|  |
| --- |
| L = [x \* x for x in range(10)]  print(L)  [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]  g = (x \* x for x in range(10))  print(g)  <generator object <genexpr> at 0x1022ef630>  print(next(g))  print(next(g))  print(next(g))  print(next(g))  print(next(g))  print(next(g)) |

我们可以直接打印出list的所有元素，但是只能一个一个打印generator的每一个元素。

使用next（generator）来获得generator的下一个返回值。

到最后一个元素之后，会抛出StopIteration错误。

改进：(for循环，因为generator也是可迭代对象)

|  |
| --- |
| g = (x \* x for x in range(10))  for n in g:  print(n)  #到最后也不会抛出错误 |

复杂的生成器：

|  |
| --- |
| def fib(max):  n, a, b = 0, 0, 1  while n < max:  yield b  a, b = b, a + b  n = n + 1  return 'done' |

若一个函数中包含了yield关键字，那么这个函数就是一个generator

|  |
| --- |
| f = fib(5)  print(f)  <generator object fib at 0x104feaaa0> |

函数是顺序执行，遇到return或者最后一行语句就返回；而变成generator的函数，每次调用next()的时候执行，遇到yield语句返回，再次执行时从上次返回的yield语句处继续执行。

|  |
| --- |
| 使用for循环来获取返回值：  for n in fib(5):  print(n) |

此时发现拿不到generator的return的返回值。若想拿到返回值，必须捕获StopIteration错误，返回值存在StopIteration的value值中：

|  |
| --- |
| while True:  try:  print(next(fib))  except StopIteration as e:  print('return :'+e.value)  break |

杨辉三角生成器：

|  |
| --- |
| 标准答案：  def triangel(n):  L=[1]  while True:  yield L  L = [L[x]+L[x+1] for x in range(len(L)-1)]  L.insert(0,1)  L.append(1)  if len(L)>n:  break  a=triangel(10)  for n in a:  print(n) |

我的答案：

|  |
| --- |
| def triangel(n):  L=[1]  i=0  while i<n:  yield L  j=0  K=[]  while j<len(L)-1:  a=L[j]+L[j+1]  K.append(a)  j=j+1  K.insert(0, 1)  K.append(1)  L=K  i=i+1  a=triangel(10)  for n in a:  print(n) |

### 字符串

##### 连接

str1 + str2 两个字符串用加号连接

‘,’.join(list) 将列表的元素用，连接用join

##### 字符串切片

str = ‘Today is a good day’

str[0:3] 表示截取字符串第一到第三个字符 与list切片类似

##### 字符串相乘

str = '-'\*20 表示20个‘-’相连接

打印：--------------------

##### 字符串分割

phone='400-800-800-1234'

print(phone.split('-'))

>>['400', '800', '800', '1234']

复杂的分割：

phone='hello world; python, I ,like, it'

import re

print(re.split('[;,]\s\*',phone)) 正则匹配

##### 字符串开头和结尾处理

filename='trace.h'

print(filename.endswith('e.h')) 字符串以什么结束

print(filename.startswith('tra')) 字符串以什么开始

##### 字符串查找和匹配

find在字符串里查找子字符串，返回子字符串所在位置的索引，找不到返回-1

title = 'python is a good language'

print(title.find('a good')) 返回10

复杂匹配：

|  |
| --- |
| import re  mydate = '11/27/2016'  if re.match('\d+\/\d+\/\d+',mydate): #正则匹配，匹配返回True，不匹配返回False  print('match')  else:  print('not match') |

##### 字符串替换

普通：str.replace(原子串，替换子串)

title = 'python is a good language'

print(title.replace('a good','a bad'))

复杂：re.sub(正则，替换子串，原字符串)

import re

content = 'boys 90,girls 200'

print(re.sub('\d+','100',content)) # boys 100,girls 100

##### 去除字符串两端的空格、制表符、换行符

str.strip()

注意：只能去除两侧的，不能去除中间的，去除中间的要用re模块

|  |
| --- |
| spam = 'SpamSpamBaconSpamEggsSpamSpam'  s = spam.strip('Samp') #将字符串两边的S、a、m、p字符全部过滤（字符的顺序无所谓）  print(s)  输出：BaconSpamEggs |

##### is和==的不同

is用来比较两个对象的内存空间是否一样，是不是用的同一块空间地址

== 比较两个对象的内容是否相等

##### 字符串连接(join +)

join的执行效率比+号高

若是连接大规模的字符串，比如要连接10万左右的字符串的时候，join的方法效率就会快很多(甚至相差百倍)

若要连接字符串:S1+S2+S3+....+SN,由于字符串是不可变的对象,执行一次就要申请一块新的内存，这样的话在N个字符串连接的过程中，会产生N-1个中间结果,每产生一个中间结果就要申请一次内存，这样会严重影响执行效率.

而join不一样，它是一次性申请总的内存，然后把字符串里面的每一个元素复制到内存中去，所以join会快很多.

|  |
| --- |
| 移除句中所有的元音字母：  sentence = 'Your mother was a hamster'  vowels = 'aeiou'  nonvowels = ''.join([l for l in sentence if not l in vowels])  上面用到了列表生成式  但是使用生成器会更好：  nonvowels = ''.join(l for l in sentence if not l in vowels)  把[]去掉，即是生成器。  join函数接收任何可迭代的数据，相对于之前把所有条目包装成一个列表,生成器在我们遍历时才产生相应的条目（join函数实际上是循环遍历拼接字符串的，在join循环的时候，后面的生成器才产生相应的数据，直接拼接到字符串上；而列表生成器是先把所有的数据组合成一个list放到内存上，然后再循环拼接字符串）。这可以使我们不必保存整个列表到内存，并且这对于处理大量数据更有效率。 |

##### 字符串格式化

print ("我叫 %s 今年 %d 岁!" % ('小明', 10))

python中的格式化符号：

|  |  |
| --- | --- |
| **符   号** | **描述** |
| %c | 格式化字符及其ASCII码 |
| %s | 格式化字符串 |
| %d | 格式化整数 |
| %u | 格式化无符号整型 |
| %f | 格式化浮点数字，可指定小数点后的精度 |

辅助符号：

|  |  |
| --- | --- |
| (var) | 映射变量(字典参数) |
| m.n. | m 是显示的最小总宽度,n 是小数点后的位数(如果可用的话) |
| 0 | 显示的数字前面填充'0'而不是默认的空格 |

‘%(name)s’ % ({‘name’:’john’}) 输出‘john’

‘%6.2f’ % 23.2352 输出： []23.23 []表示一个空格，算上小数点和小数不满6，在最左边补空格

'%07.2f' % 23.2352 输出：0023.24 不够长度填充0

format格式化：

|  |
| --- |
| >>>"{} {}".format("hello", "world") # 不设置指定位置，按默认顺序  'hello world'    >>> "{0} {1}".format("hello", "world") # 设置指定位置  'hello world'    >>> "{1} {0} {1}".format("hello", "world") # 设置指定位置  'world hello world'  print("网站名：{name}, 地址 {url}".format(name="菜鸟教程", url="www.runoob.com"))    # 通过字典设置参数  site = {"name": "菜鸟教程", "url": "www.runoob.com"}  print("网站名：{name}, 地址 {url}".format(\*\*site))    # 通过列表索引设置参数  my\_list = ['菜鸟教程', 'www.runoob.com']  print("网站名：{0[0]}, 地址 {0[1]}".format(my\_list)) # "0" 是必须的 |

##### 字节字符串（bytes）和文本字符串（string）区别

在python中：

|  |
| --- |
| b'haha' # 字节字符串  'haha' # 文本字符串 |

文本字符串str是字符序列，它是一种抽象的概念，不能直接存储在硬盘；字节字符串是字节序列，它可以直接存储在硬盘。它们之间的映射被称为编码/解码。在Python中，程序中的文本都用文本字符串（str）表示。

所以你在读取文件时，需要把字节序列解码成utf-8字符串；你在写入文件时，需要把utf-8字符串编码成字节序列。

|  |
| --- |
| # 解码  >>> b'haha'.decode('ascii')  'haha'  # 编码  >>> 'haha'.encode()  b'haha' |

##### 字符串内建函数

str.islower()全是小写 返回True

str.isupper()全是大写，返回True

str.capitalize() 首字母大写，其他字母小写

str.casefold() 全部转为小写

str.count(sub[,start[,end]] sub子串在str中出现的次数，start和end为搜索的开始结束位置，不填默认从头搜到尾

以下三个会产生新的字符串，需要赋值给变量，并不能改变原来的字符串

new = str.center(num,fillchar) 字符串居中，左右填充fillchar，填充到num个字符

new = str.rjust(num[,fillchar]) 字符串右对齐，填充至num个字符（fillchar），默认填充空格。

new = str.ljust(num[,fillchar]) 字符串左对齐，同上

str.isalnum() 检测字符串是否由字母和数字组成,成功返回true，失败返回false

str.isalpha() 检测字符串是否只由字母组成，成功返回true，失败返回false

str.isdigit() 检测字符串是否只由数字组成，成功返回true，失败返回false

str.isnumeric()检测字符串是否只由数字组成，成功返回true，失败返回false

str.lower() 转小写

str.upper()转大写

str.title() 每个单词首字母都转为大写

### json

python中用json模块来操作json

import json

json.dumps(): 对数据进行编码。

json.loads(): 对数据进行解码。

若处理的是文件而不是字符串，使用json.dump()和json.load()来解编码json数据

|  |
| --- |
| # 写入 JSON 数据  with open('data.json', 'w') as f:  json.dump(data, f)  # 读取数据  with open('data.json', 'r') as f:  data = json.load(f) |

### 文件、目录

##### 文件处理

import os

若在当前目录下有abc.txt

重命名： os.rename('abc.txt','a123.txt') #重命名 文件和文件名

删除文件：

if os.path.exists(‘abc.txt’): #判断文件是否存在

os.remove(‘abc.txt’) #删除文件

##### 文件路径

path='/users/Python/Data/info.txt'

获取所在目录：

os.path.dirname(path) # /users/Python/Data

获取文件名：

os.path.basename(path) #info.txt

分割目录和文件名：（返回一个tuple）

os.path.split(path) # ('/users/Python/Data', 'info.txt')

合并目录和文件名：

os.path.join(dirname,filename) # dirname\filename

分割文件名和扩展名：

os.path.splitext(filename) # ('learning', '.py')

##### 创建并访问目录

创建目录：

|  |
| --- |
| if not os.path.exists('test01'): #先判断目录是否存在，否则会报错  os.mkdir('test01') |

罗列当前目录下的所有文件和文件夹：（只会显示当前目录一层的）

os.listdir('.') # ['learning.py', 'main.py', 'test1']

生成当前目录下的所有文件和目录：（会往下层层查找）

list(os.walk('.')) #[('.', ['test1'], ['learning.py', 'main.py']), ('.\\test1', [], ['test.txt'])]

|  |
| --- |
| os.walk()获得的是个生成器generator。  a代表当前目录名，b代表子目录名，c代表当前目录下的文件名  一层一层向内查找  for a,b,c in os.walk('.'):  #python中in后面若是这样的形式：[(x,y,z),(e,f,g)] ，则for后面可以直接用a,b,c跟x,y,z对应起来（类比for index,val in enumerate(list)）  print(a)  for x in b:  print(' '+x)  for y in c:  print(' '+ y) |

返回当前目录：

os.getcwd()

删除目录：

os.rmdir(‘test1’) #只能删除空目录

##### 判断是否是文件或目录

os.path.isfile(‘test1’) 是否是文件

os.path.isdir(‘test1’) 是否是目录

os.path.exists(‘test1’) 判断文件或目录是否存在

##### shutil模块

复制文件和文件夹：

shutil.copy(source,destination) 将source的文件复制到destination处的文件夹，若destination是个文件名，它将作为被复制文件的新名字。

返回：被复制后的文件路径

shutil.copytree(source,destination) 复制整个文件夹，以及它包含的文件夹和文件

返回：新文件夹的路径

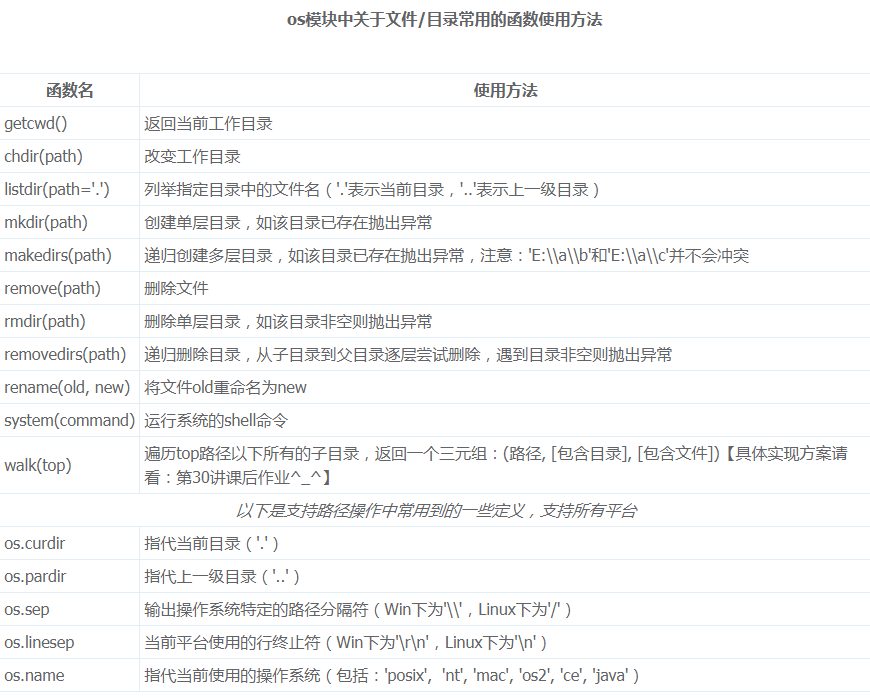
shutil.move(source, destination) 将source处的文件移动到destination中，并保持原来的文件名；若destination是文件名，则移动后改为该名

返回：移动后文件的路径

注意：若目标文件夹不存在会报错

shutil.rmtree(path) 删除path下的所有文件和文件夹 （永久删除，慎用！！）

##### 方法汇总





os.path.dirname(path) 返回最后一个斜杠前面的内容，无论最后一个斜杠后面是文件名还是目录名

os.path.basename(path) 返回最后一个斜杠后面的内容，无论此内容是文件名还是目录名

os.path.split(path) 以最后一个斜杠为分界分割

os.path.sep 文件夹分割斜杠

os.path.abspath(path) 返回绝对路径

os.path.relpath(path,start) 返回path相对于start的相对路径

### 文件处理

##### 文件打开

f =open(filename,mode='r',buffering=-1) 获得一个File对象

#mode可以是读，写或者追加,一般默认是读文件

#buffering主要是设置缓存区,一般我们都是用-1表示用系统的默认缓冲区,若写100,表示缓冲区为100

注意：windows下新建一个文件的默认编码是gbk，可以这样：

f = open("out.html","w",encoding='utf-8')

##### 文件读取

f.readline()#表示读文件的一行

f.readlines()#表示把文件从头到尾都读出来，并保存为一个列表，一行为一个元素

f.read([n]) #读n个字符 ，参数省略则读全部

还可以：

f = open(‘test.txt’)

for line in f:

print(line)

还可以将文件对象的数据存放进list：

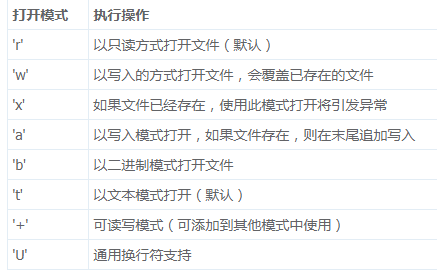
f = open(‘test.txt’)

list(f)

##### 文件写入

f.write(str)#表示把字符串写入

f.writelines(list)#表示把一个列表写入



##### 文件关闭

f.close() #容易忘

with方法会自动关闭文件

with open(r'somefile.txt') as f:

f.readlines()

##### 文件的读取位置

f.seek(offset,whence=0)

#seek主要是在文件中移动指针,从whence(0表示文件头,1表示当前位置,2表示文件尾)偏移offset个字节

f.tell() 返回当前指针的位置

##### 文件处理高级用法

1)

fileinput 模块允许你循环一个或多个文本文件的内容,非常实用

|  |
| --- |
| import fileinput  import glob  #利用glob模块过滤出当前目录下所有的txt文件  #inplace=1 标准输出,比如print会被重定向到打开文件  #把每一行去掉回车，然后把java替换为Python  for line in fileinput.input(glob.glob('\*.txt'),inplace=1):  print line.strip().replace('Java','Python')  fileinput.close()#关闭文件句柄 |

2)

with:

不用with时：

|  |
| --- |
| f = open("123.txt")  try:  data = f.read()  print int(data)  finally:  print 'file close'  f.close() |

若改成with：

|  |
| --- |
| with open('123.txt') as f:  line=f.read()  print line |

内部原理：

with context as f:

with\_suite

1. 计算context的值，返回一个对象，这个对象是一定要支持上下文管理器。一定要包含\_\_enter\_\_()和\_\_exit\_\_()方法
2. 进入上下文管理器:调用上下文管理器的\_\_enter\_\_()方法,返回运行时的上下文相关的对象,with语句会把这个返回值绑定到目标对象,也就是获得文件句柄，并把这个句柄赋值给f
3. 执行with中的代码
4. 如果with中的代码顺利执行完毕，调用上下文管理器里的\_\_exit()\_\_方法,其返回值直接忽略
5. 若在with中的代码执行过程中，发生了异常，也会调用上下文管理器的\_\_exit\_\_()方法,并把异常的类型，异常的值和traceback信息全部都以参数的形式传递给\_\_exit\_\_()方法。

\_\_exit\_\_(exception\_type,exception\_value,traceback),这个方法可以处理异常，清理现场,处理with块语句执行完要处理的动作

##### 对象持久性

透明地存储 Python 对象，而不丢失其身份和类型等信息，需要某种形式的对象序列化。它是一个将任意复杂的对象转成对象的文本或二进制表示的过程

在 Python 中，这种序列化过程称为 pickle，可以将对象 pickle 成字符串、磁盘上的文件或者任何类似于文件的对象，也可以将这些字符串、文件或任何类似于文件的对象 unpickle 成原来的对象。

|  |
| --- |
| import pickle  #将对象保存为二进制文件  list = [1,2,3,4,5,6,7,8,9]  f = open('test.pkl','wb') 一定要用wb模式打开  pickle.dump(list,f)  f.close()  #将对象从二进制文件中读取  f=open('test.pkl','rb') 一定要用rb模式打开文件  list = pickle.load(f)  print(list) |

### 异常处理

##### 基本语法

第一种：

try:

do something

except Exception as e:

print(e) #e输出的是错误信息

else:

pass

#try捕获异常，Exception是异常的种类，e表示异常的信息

若有异常，走except，若无异常，走else

第二种：

try:

do something

except Exception:

handle error

finally:

do finally

#无论有没有异常，都会执行finally

若分不清哪种错误类型，直接except Exception即可；

##### 错误类型

NameError 变量或函数名拼写错误

AttributeError访问一些未知的对象属性

|  |
| --- |
| line='Python is easy'  print line.upperr()  >>AttributeError: 'str' object has no attribute 'upperr' |

IndexError 列表越界（超过了最大索引）

SyntaxError忘记在if/for/while/def 声明末尾添加 冒号

IOError 打开一个不存在的文件名

##### 主动抛出异常

用raise语句,抛出异常之后，若没有try/except的话就会扔给python解释器去处理

raise Exception(错误信息)

##### 用assert断言来抛出异常

assert expression,args

expression:表达式

args：对条件判断的描述信息

相当于在代码里设置了个判断语句，当不满足expression（即expression为False）时，抛出异常信息args

### 常见问题

##### Python UnicodeEncodeError: 'gbk' codec can't encode character

写文件时：

|  |
| --- |
| f.write(txt) |

txt是一个字符串，它是通过decode解码过的字符串，关键点就要来了：目标文件的编码是导致标题所指问题的罪魁祸首。如果我们打开一个文件：

|  |
| --- |
| f = open("out.html","w") |

在windows下面，新文件的默认编码是gbk，这样的话，python解释器会用gbk编码去解析我们的网络数据流txt，然而txt此时已经是decode过的unicode编码，这样的话就会导致解析不了，出现问题

解决：

|  |
| --- |
| f = open("out.html","w",encoding='utf-8') |

##### 关于可变和不可变对象

python中string，tuple，number是不可变对象，list和dict是可变对象

表面上是这样定义的，但是注意：

前三种，给他们重新赋值时，是在内存中重新建立了新对象，变量名指向了这个新对象，使用id(var)可以实时监测内存地址的变化。

list和dict的可变，只是说他内部元素是可变的；若给他们重新赋值，他的内存地址也会改变，这种情况下他们也是不可变的

|  |
| --- |
| a=[1,2]  print(id(a))  def fn(a): #这里传递过来的a参数，其实是全局变量a的一个引用，指向的和全局变量是一个内存地址  print(id(a))  a.append(2)  print(id(a))  fn(a)  print(id(a)) |

输出：18078344

18078344

18078344

18078344

|  |
| --- |
| a=[1,2]  print(id(a))  def fn(a): #参数a指向的也是[1,2]的内存地址，只不过在下面重新赋值了，就指向了一个新的内存地址，这里跟string、number、tuple是一样的  print(id(a))  a=[3,4]  print(id(a))  fn(a)  print(id(a)) |

输出：18339400

18339400

18340360

18339400

##### global

|  |
| --- |
| x=12  def fn():  '我是一个函数文档'  global x #使用了global关键字，x就是全局变量了，并不是他的一个引用，所以可以在函数内直接修改全局变量  x=11  print('输出',x)  fn() |

##### nonlocal

修改外层非全局作用域下的变量，可以用nonlocal

|  |
| --- |
| def outer():  num = 10  def inner():  nonlocal num # nonlocal关键字声明  num = 100  print(num)  inner()  print(num)  outer() |

##### python中的拷贝

a=[1,2,3]  
b=a  
b.append(4)  
print id(a),a  
print id(b),b

>>

39161288 [1, 2, 3, 4]

39161288 [1, 2, 3, 4]

用id()函数来查看对象的唯一识别号,发现a,b是一样的,原因是因为b只是a的引用，都是同一个地址，并没有实现真正的copy

注意：这里跟php有所不同，php里若b的值改变了，则意味着b不再指向a指向的内存地址了，而是指向新值得内存地址；但是在python中，b的值改变了，还是指向同一个地址，因此a的值也会跟着改变

如果你想修改一个对象，但是又不需要改动原对象,必须要引入copy模块

import copy  
a=[1,2,3]  
b=copy.copy(a)  
b.append(4)  
print 'a:',a  
print 'b:',b

>>

a: [1, 2, 3]

b: [1, 2, 3, 4]

##### 关于深拷贝和浅拷贝

<http://python.jobbole.com/82294/>

浅拷贝是复制了一个对象到一个内存地址，但是对象里的元素还是使用原来的内存地址。

因为python中字符串是不可变的，所以如果改变字符串元素的值，则会重新分配一个内存地址，而list是可变的，若改变一个list元素的值，则list元素的内存地址不会变。

深拷贝不仅将对象复制到一个新内存地址上，而且给里面的list元素也重新分配了新的内存地址，所以改变备份的list值并不会影响原来的list值

### 内置函数

chr(i) i是ASKII码，返回该ASKII对应的字符 0<=i<=255

property():

x = property(getSize,setSize,delSize)

|  |
| --- |
| class C(object):  def \_\_init\_\_(self,size=10):  self.size = size  def getSize(self):  return self.size  def setSize(self,value):  self.size = value  def delSize(self):  del self.size  x = property(getSize,setSize,delSize)  c = C()  print(c.x)  c.x = 3  print(c.x)  del c.x  print(c.x) |

如果c是C的实例:

c.x将触发getSize;

c.x=3将处罚setSize;

del c.x将触发delSize

property其实是使用了描述符：

下面模仿property，创造一个MyProperty类

|  |
| --- |
| class MyProperty(object):  def \_\_init\_\_(self,fget,fset,fdel):  self.fget = fget  self.fset = fset  self.fdel = fdel  def \_\_get\_\_(self,instance,owner): #定义当描述符的值被读取时的行为  return self.fget(instance)  def \_\_set\_\_(self,instance,value): #定义当描述符的值被改变时的行为  self.fset(instance,value)  def \_\_delete\_\_(self, instance): #定义当描述符的值被删除时的行为  self.fdel(instance)  class C(object):  def \_\_init\_\_(self,x):  self.x = x  def getx(self):  return self.x  def setx(self,value):  self.x = value  def delx(self):  del self.x  y = MyProperty(getx,setx,delx)  c = C(3)  print(c.y) #获取描述符  c.y = 123 #改变描述符  print(c.y)  print(c.x)  del c.y #删除描述符  print(c.y) |

abs() 绝对值

round(x[,n]) 四舍五入 n为保留几位小数

### pycharm快捷键

Alt+Enter :当使用一个未引用的模块时，可以用它快捷引入

### 批量运行python文件

python文件第一行以#！开始，告诉计算机你想让python来运行这个程序

在 Windows 上，第一行是 #! python3。

 在 OS X，第一行是 #! /usr/bin/env python3。

 在 Linux 上，第一行是 #! /usr/bin/python3。

window里python解释器在他的安装目录下的python.exe,或者方便的 py.exe 程序将读取.py 文件源代码顶部的#!行，并针对该脚本运行相应的 Python版本。

可以创建一个py.bat批量处理文件，用py.exe来运行python程序。

py.bat文件如下：

|  |
| --- |
| @py.exe F:\python\osram\test4\pw.py %\*  @pause |

将py.bat 文件所在的文件夹添加到环境变量，这样你可以直接在命令行输入文件名即可运行文件。

命令行参数存储在sys.argv中，sys.argv列表的第一项总是运行的文件名。第二项是第一个命令行参数，用sys.argv[1]获取

cmd命令行里运行：py.bat 参数 即可，此参数可用sys.argv[1]获得