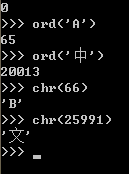
**Python基础**

**(一)字符串和编码**

1. **对于单个字符的编码**

1>ord()函数获取字符的整数表示;

2>chr()函数把编码转换为对应的字符;



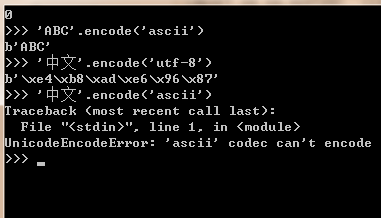
1. **Python的字符串类型是str,在内存中以Unicode表示**,一个字符对应若干个字节.如果要在网络上传输,或者保存到磁盘上,就需要把str变为以字节为单位的bytes.

Python对bytes类型的数据用带b前缀的单引号或双引号表示:

X=b’ABC’

要注意区分’ABC’和b’ABC’,前者是str,后者虽然内容显示得和前者一样,但bytes的每个字符都只占用一个字节.

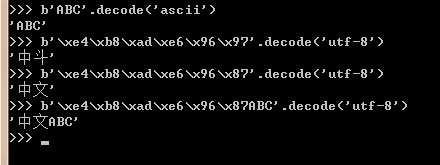
以Unicode表示的str 同过encode()方法可以编码为指定的bytes,例如:



纯英文的str可以用ASCII编码为bytes，内容是一样的，含有中文的str可以用UTF-8编码为bytes。含有中文的str无法用ASCII编码，因为中文编码的范围超过了ASCII编码的范围，Python会报错。

在bytes中，无法显示为ASCII字符的字节，用\x##显示。

1. **如果我们从网络或磁盘上读取了字节流,那么读到的数据就是bytes.要把bytes 变为str,就需要decode()方法:**



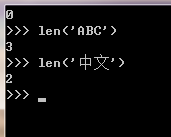
如果bytes中包含无法解码的字节,decode()方法就会报错:

如果bytes中只有一小部分无效的字节,可以传入errors=’ignore’忽略错误的字节:

>>> b'\xe4\xb8\xad\xff'.decode('utf-8', errors='ignore')

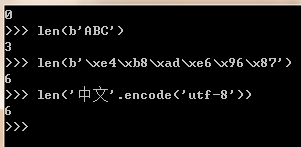
'中'

1. **要计算str 包含多少这个字符,可以用len()函数:**



Len()函数计算的是str的字符数.如果换成bytes,len()函数就计算

字节数



可见,1个中文字符经过UTF-8编码后通常会占用3个字节,而1个英文字符只占用一个字节.

在操作字符串时,我们经常遇到str和bytes的相互转换.为了避免乱码问题,应当始终坚持使用utf-8编码对str和bytes进行转换.\

由于Python源代码也是一个文本文件,所以当你的源代码中包含中文的时候,在保存源代码时,就需要务必指定保存为UTF-8编码.当Python解释器读取源代码时,为了让它按UTF-8编码读取,我们通常在文件开头写上这两行.

#!/usr/bin/env Python3

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

第一行注释是为了告诉Linux/OS X系统，这是一个Python可执行程序，Windows系统会忽略这个注释；

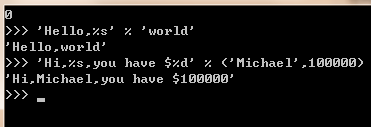
第二行注释是为了告诉Python解释器，按照UTF-8编码读取源代码，否则，你在源代码中写的中文输出可能会有乱码。

申明了UTF-8编码并不意味着你的.py文件就是UTF-8编码的，必须并且要确保文本编辑器正在使用UTF-8 without BOM编码

1. **格式化**

最后一个常见的问题是如何输出格式化的字符串。我们经常会输出类似'亲爱的xxx你好！你xx月的话费是xx，余额是xx'之类的字符串，而xxx的内容都是根据变量变化的，所以，需要一种简便的格式化字符串的方式。

在Python中采用的格式化方式和C语言是一致的,用 % 实现

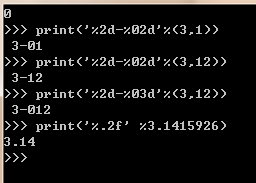


你可能猜到了,%运算符就是用来格式化字符串的.在字符串内部,%s 表示用字符串替换,%d表示用整数替换,有几个%?占位符,后面就跟几个变量或者值,顺序要对应好.如果只有一个%?,括号刻印省略.

常见的占位符有:

|  |  |
| --- | --- |
| **占位符** | **替换内容** |
| **%d** | **整数** |
| **%f** | **浮点数** |
| **%s** | **字符串** |
| **%x** | **十六进制整数** |

其中,格式化整数和浮点数还可以指定是否补0和整数与小数的位数:



如果你不确定应该用什么,%s永远起作用,它会把任何数据类型转换为字符串:

>>> 'Age: %s. Gender: %s' % (25, True)

'Age: 25. Gender: True'

有些时候,字符串里面的%是一个普通字符怎么办?这个时候就需要转义,用%%来表示一个%:

>>> 'growth rate: %d %%' % 7'

growth rate: 7 %'

**Format()**

另一种格式化字符串的方式是使用字符串的format()方法 , 它会用传入的参数依次替换字符串内的占位符{0},{1}....... ,不过这种方式写起来比%要麻烦的多:

>>> 'Hello, {0}, 成绩提升了 {1:.1f}%'.format('小明', 17.125)'

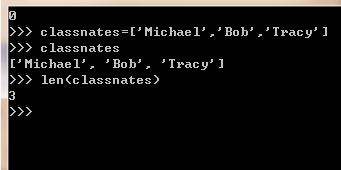
Hello, 小明, 成绩提升了 17.1%'

1. **使用list 和 tuple**
2. **List**

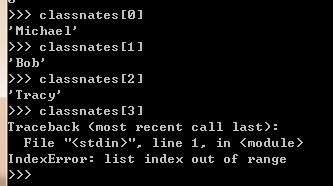
Python内置的一种数据类型是列表:list. List是一种有序的集合.可以随时添加和删除期中的元素.

Classmates = [‘Michael’,’Bob’,’Tracy’]

变量classmates就是一个list.用len()函数可以获得list元素的个数:



用索引来访问list中每一个未知的元素,记得索引是从0开始的.



当索引超出了范围时,Python会报一个indexError 错误,所以,

要确保索引不要越界,记得最后一个元素的索引是 len(classnates) -1.

如果要取最后一个元素,除了计算索引位置外,还可以用-1 做索引,直接获取最后一个元素:

>>> classmates[-2]

'Bob'

>>> classmates[-3]

'Michael'

>>> classmates[-4]

Traceback (most recent call last):

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

IndexError: list index out of range

当然,倒数第4个就越界了.

List是一个可变的有序表,所以可以往list中追加元素到末尾:

>>> classmates.append('Adam')

>>> classmates

['Michael', 'Bob', 'Tracy', 'Adam']

也可以把元素插入到指定的位置,比如索引号为1 的位置:

>>> classmates.insert(1, 'Jack')>>> classmates

['Michael', 'Jack', 'Bob', 'Tracy', 'Adam']

要删除list末尾的元素,用 pop()方法:

>>> classmates.pop()'Adam'>>> classmates

['Michael', 'Jack', 'Bob', 'Tracy']

要删除指定位置的元素,用pop(i)方法,其中i是索引位置

>>> classmates.pop(1)'Jack'>>> classmates

['Michael', 'Bob', 'Tracy']

要把某个元素替换成别的元素,可以直接赋值给对应的索引位置:

>>> classmates[1] = 'Sarah'>>> classmates

['Michael', 'Sarah', 'Tracy']

List里面的元素的数据类型也可以不同,

List元素也可以是另一个list.

>>> s = ['python', 'java', ['asp', 'php'], 'scheme']>>> len(s)4

要注意s只有4个元素，其中s[2]又是一个list

要拿到’PHP’可以写s[2][1],因此s可以看成是一个二维数组,类似的还有三维数组,四维......数组 不过很少用到.

如果一个list中一个元素也没有,就是一个空的list,它的长度为0;

1. **Tuple**

另一种有序列表叫元组:tuple.tuple和list非常类似.但是tuple一旦初始化就不能修改,比如同样是列出同学的名字:

>>> classmates = ('Michael', 'Bob', 'Tracy')

现在,classmates这个tuple不能变了,它也没有append(),insert()这样的方法.其他获取元素的方法和list是一样的,你可以正常地使用 classmates[0],classmates[-1],但不能赋值成另外的元素.

问题:不可变的tuple有什么意义?

因为tuple不可变,所以代码更安全.如果可能,能用tuple代替list就尽量用tuple.

Tuple的陷阱:当比定义一个tuple时,在定义的时候,tuple元素就必须被确定下来,比如:

>>> t = (1, 2)>>> t

(1, 2)

如果要定义一个空的tuple,可以写成():

>>> t = ()

>>> t

()

但是,要定义一个只有1个元素的tuple,定义时必须加一个逗号, 来消除歧义.

>>> t = (1,)>>> t

(1,)

Python在显示只有1个元素的tuple时,也会加一个逗号,以免你误解成数学计算意义上的括号.

最后来看一个”可变的”tuple:

>>> t = ('a', 'b', ['A', 'B'])>>> t[2][0] = 'X'>>> t[2][1] = 'Y'>>> t

('a', 'b', ['X', 'Y'])

这个tuple定义的时候有三个元素,分别是’a’,’b’和一个list.不是说tuple一旦被定以后就不可变了吗?怎么后来又变了?

表面上看,tuple的元素确实变了,但其实变的不是tuple的元素,而是list的元素.tuple一开始指向的list并没有改变成变的list,所以,tuple所谓的”不变”是说,tuple的每个元素,指向永远不变.即指向’a’,就不能改成指向’b’,指向一个list,就不能改成指向其他对象,但指向的这个list本身是可变的!

理解了’指向不变后’,要创建一个内容也不变的tuple怎么做?那就必须保证tuple的每一个元素本身也不能变.

1. **条件判断**

计算机之所以能做很多自动化的任务,因为它可以自己做条件判断.

在Python程序中,用if语句实现判断:

age = 20

**if** age >= 18:

**print**('your age is', age)

**print**('adult')

根据Python的缩进规则,如果if语句判断是”True”,就把缩进的两个print语句执行了.否则,什么也不做.也可以给 if 添加一个else 语句,意思是,如果if判断是False,不要执行if的内容,去把else执行了:

age = 3**if** age >= 18:

**print**('your age is', age)

**print**('adult')**else**:

**print**('your age is', age)

**print**('teenager')

注意不要少写了冒号:

当然上面的判断是很粗略的,完全可以用elif 做更细致的判断.

age = 3**if** age >= 18:

print('adult')**elif** age >= 6:

print('teenager')**else**:

print('kid')

elif是else if的缩写,完全可以有多个elif,所以if

语句的完整形式就是:

if <条件判断1>:

<执行1>

elif <条件判断2>:

<执行2>

elif <条件判断3>:

<执行3>

else:

<执行4>

If语句执行有个特点,它是从上往下判断,如果在某个判断上是True,把该判断对应额语句执行后,就忽略剩下的elif和 else.

If判断条件还可以简写,比如写:

If x:

Print(‘True’)

只要 X 是非零数值,非空字符串,非空list等,就判断为True,否则为False.

**再议input**

最后看一个有问题的条件判断.很多同学会用input()读取用户的输入,这样可以自己输入,程序运行得更有意思:

birth = input('birth: ')

**if** birth < 2000:

**print**('00前')

**else**:

**print**('00后')

输入1982，结果报错：

这是因为input()返回的数据类型是str,str不能直接和整数比较,必须先把str转换成整数.Python提供了 int()函数来完成这件事情

s = input('birth: ')

birth = int(s)

**if** birth < 2000:

**print**('00前')

**else**:

**print**('00后')

1. **循环**

为了让计算机能计算成千上万次的重复运算,我们就需要循环语句.

Python的循环有两种,一种是 for...in循环,依次把list或tuple中的每个元素迭代出来,看例子:

names = ['Michael', 'Bob', 'Tracy']**for** name **in** names:

print(name)

执行这段代码,会依次打印 names的每一个元素:

所以 for x in ... 循环就是把每个元素带入变量 X,然后执行缩进块的语句.

如果要计算 1-100的整数之和,从1写到100有点困难,幸好Python提供一个range()函数,可以生出一个整数序列,再通过list()函数可以转换为list.比如 range(5)生成的序列是从0开始小于5的整数:

>>> list(range(5))

[0, 1, 2, 3, 4]

**Break**

再循环中,break语句可以提前退出循环.例如,本来要循环打印1-100的数字,如果要提前结束循环,可以用break语句

n = 1**while** n <= 100:

**if** n > 10: *# 当n = 11时，条件满足，执行break语句*

**break** *# break语句会结束当前循环*

**print**(n)

n = n + 1**print**('END')

执行上面的代码可以看到,打印出1-10后,紧接着打印 END,程序结束.可见break 的作用是提前结束循环.

**Continue**

在循环过程中,也可以通过 continue语句,跳过当前的这次循环,直接开始下一次循环.

比如我们只想打印1-100的奇数,可以用continue语句跳过某些循环

n = 0**while** n < 10:

n = n + 1

**if** n % 2 == 0: *# 如果n是偶数，执行continue语句*

**continue** *# continue语句会直接继续下一轮循环，后续的print()语句不会执行*

**print**(n)

continue的作用是提前结束本轮循环，并直接开始下一轮循环。

**小结**

循环是计算机做重复任务的有效的办法.

Break 语句可以在循环过程中直接退出循环,而从continue 语句可以提前结束本轮循环,并直接开始下一轮循环.这两个语句通常都必须配合if语句使用.

要特别注意,不要滥用 break 和 continue语句. break和 continue会造成代码执行逻辑分叉过多,容易出错.

有些时候,如果代码写的有问题,会让程序陷入’死循环’,也就是永远循环下去.这时可以用 Ctrl+c 退出程序,或者强制结束Python进程.

1. **使用dict 和 set**

**dict**

Python 内置了字典: dict的支持,dict全称dictionary,在其他语言中也称为 map .使用键-值(key-value)存储,具有极快的查找速度.

举个例子.假设要根据同学的名字查找对应的成绩.如果用list实现,需要两个list :

names = ['Michael', 'Bob', 'Tracy']

scores = [95, 75, 85]

给定一个名字,要查找对应的成绩,就要先在names中找到对应的位置,再从scores中取出对应的成绩,list越长,耗时越长.

如果用dict实现,只需要一个’名字’-’成绩’的对照表,直接根据名字查找成绩,无论这个表有多大.查找速度不会变慢.

用Python写一个dict如下:

>>> d = {'Michael': 95, 'Bob': 75, 'Tracy': 85}

>>> d['Michael']

95

为什么dict查找速度这么快?因为dict的实现原理和查字典是一样的.假设字典包含了1万个汉字,我们要查某一个字,一个办法是把字典从第一页往后翻.直到找到我们想要的字为止,这种方法就是在list中查找元素的方法.list越大,查找越慢.

第二种方法是先在字典的索引表里(比如部首表)查这个字对应的页码,然后直接翻到该页,找到这个字.无论找哪个字,这种查找速度都非常快,不会随着字典大小的增加而变慢.

大数据放入dict的方法,除了初始化时指定外,还可以通过可以放入:

>>> d['Adam'] = 67

>>> d['Adam']

67

由于一个key只能对应一个value,所以,多次对一个key放入value,后面的值会把前面的值冲掉:

>>> d['Jack'] = 90

>>> d['Jack']

90

>>> d['Jack'] = 88

>>> d['Jack']

88

如果可以不存在,dict就会报错:

要避免key不存在的错误,有两种办法,一是通过 in

判断key是否存在:

>>> 'Thomas' **in** d

False

二是通过dict提供的get()方法,如果key不存在,可以返回None,或者自己指定的value:

>>> d.get('Thomas')

>>> d.get('Thomas', -1)

-1

注意:返回 None的时候Python的交互环境不显示结果.

要删除一个key,用pop(key)方法,对应的value也会从dict中删除:

>>> d.pop('Bob')

75

>>> d

{'Michael': 95, 'Tracy': 85}

请务必注意,dict内部存放的顺序和key放入的顺序是没有关系的.

和list比较,dict有以下几个特点:

1. 查找和插入的速度极快,不会随着key的增加而变慢;
2. 需要占用大量的内存,内存浪费多.

而list相反

1. 查找和插入的时间随着元素的增加而增加;

占用空间小,浪费内存很少.

所以,dict是用空间来换取时间的一种方法.

dict可以用在需要高速查找的很多地方,在Python代码中几乎无处不在,正确使用dict非常重要,需要牢记的第一条就是 dict的key 必须是**不可变对象**

可是因为dict根据key来计算value的存储位置,如果每次计算相同的key得出的结果不同,那dict内部就完全混乱了.这个通过可以计算位置的算法称为哈希算法(Hash)

要保证hash的正确性,作为key的对象就不能变.在Python中,字符串,整数等都是不可变的,因此,可以放心得作为key.而list是可变的,就不能作为key

**Set**