## python 基础

Python是一种计算机编程语言。计算机编程语言和我们日常使用的自然语言有所不同，最大的区别就是，自然语言在不同的语境下有不同的理解，而计算机要根据编程语言执行任务，就必须保证编程语言写出的程序决不能有歧义，所以，任何一种编程语言都有自己的一套语法，编译器或者解释器就是负责把符合语法的程序代码转换成CPU能够执行的机器码，然后执行。Python也不例外。

Python的语法比较简单，采用缩进方式，写出来的代码就像下面的样子：

*# print absolute value of an integer:*

a = 100

**if** a >= 0:

**print**(a)

**else**:

**print**(-a)

以#开头的语句是注释，注释是给人看的，可以是任意内容，解释器会忽略掉注释。其他每一行都是一个语句，当语句以冒号:结尾时，缩进的语句视为代码块。

缩进有利有弊。好处是强迫你写出格式化的代码，但没有规定缩进是几个空格还是Tab。按照约定俗成的管理，应该始终坚持使用4个空格的缩进。

缩进的另一个好处是强迫你写出缩进较少的代码，你会倾向于把一段很长的代码拆分成若干函数，从而得到缩进较少的代码。

缩进的坏处就是“复制－粘贴”功能失效了，这是最坑爹的地方。当你重构代码时，粘贴过去的代码必须重新检查缩进是否正确。此外，IDE很难像格式化Java代码那样格式化Python代码。

最后，请务必注意，Python程序是大小写敏感的，如果写错了大小写，程序会报错。

小结

Python使用缩进来组织代码块，请务必遵守约定俗成的习惯，坚持使用4个空格的缩进。

在文本编辑器中，需要设置把Tab自动转换为4个空格，确保不混用Tab和空格。

## 数据类型与变量

### **数据类型**

计算机顾名思义就是可以做数学计算的机器，因此，计算机程序理所当然地可以处理各种数值。但是，计算机能处理的远不止数值，还可以处理文本、图形、音频、视频、网页等各种各样的数据，不同的数据，需要定义不同的数据类型。在Python中，能够直接处理的数据类型有以下几种：

#### **整数**

Python可以处理任意大小的整数，当然包括负整数，在程序中的表示方法和数学上的写法一模一样，例如：1，100，-8080，0，等等。

计算机由于使用二进制，所以，有时候用十六进制表示整数比较方便，十六进制用0x前缀和0-9，a-f表示，例如：0xff00，0xa5b4c3d2，等等。

#### **浮点数**

浮点数也就是小数，之所以称为浮点数，是因为按照科学记数法表示时，一个浮点数的小数点位置是可变的，比如，1.23x109和12.3x108是完全相等的。浮点数可以用数学写法，如1.23，3.14，-9.01，等等。但是对于很大或很小的浮点数，就必须用科学计数法表示，把10用e替代，1.23x109就是1.23e9，或者12.3e8，0.000012可以写成1.2e-5，等等。

整数和浮点数在计算机内部存储的方式是不同的，整数运算永远是精确的（除法难道也是精确的？是的！），而浮点数运算则可能会有四舍五入的误差。

#### **字符串**

字符串是以单引号'或双引号"括起来的任意文本，比如'abc'，"xyz"等等。请注意，''或""本身只是一种表示方式，不是字符串的一部分，因此，字符串'abc'只有a，b，c这3个字符。如果'本身也是一个字符，那就可以用""括起来，比如"I'm OK"包含的字符是I，'，m，空格，O，K这6个字符。

如果字符串内部既包含'又包含"怎么办？可以用转义字符\来标识，比如：

'I\'m \"OK\"!'

表示的字符串内容是：

I'm "OK"!

转义字符\可以转义很多字符，比如\n表示换行，\t表示制表符，字符\本身也要转义，所以\\表示的字符就是\，可以在Python的交互式命令行用print()打印字符串看看：

>>> **print**('I\'m ok.')

I'm ok.

>>> print('I\'m learning\nPython.')

I'm learning

Python.

>>> print('\\\n\\')

\

\

如果字符串里面有很多字符都需要转义，就需要加很多\，为了简化，Python还允许用r''表示''内部的字符串默认不转义，可以自己试试：

>>> print('\\\t\\')

\ \

>>> print(r'\\\t\\')

\\\t\\

如果字符串内部有很多换行，用\n写在一行里不好阅读，为了简化，Python允许用'''...'''的格式表示多行内容，可以自己试试：

>>> print('''line1

... line2

... line3''')

line1

line2

line3

上面是在交互式命令行内输入，注意在输入多行内容时，提示符由>>>变为...，提示你可以接着上一行输入。如果写成程序，就是：

print('''line1

line2

line3''')

多行字符串'''...'''还可以在前面加上r使用，请自行测试。

#### **布尔值**

布尔值和布尔代数的表示完全一致，一个布尔值只有True、False两种值，要么是True，要么是False，在Python中，可以直接用True、False表示布尔值（请注意大小写），也可以通过布尔运算计算出来：

>>> True

True

>>> False

False

>>> 3 > 2

True

>>> 3 > 5

False

布尔值可以用and、or和not运算。

and运算是与运算，只有所有都为True，and运算结果才是True：

>>> True **and** True

True

>>> True **and** False

False

>>> False **and** False

False

>>> 5 > 3 **and** 3 > 1

True

or运算是或运算，只要其中有一个为True，or运算结果就是True：

>>> True **or** True

True

>>> True **or** False

True

>>> False **or** False

False

>>> 5 > 3 **or** 1 > 3

True

not运算是非运算，它是一个单目运算符，把True变成False，False变成True：

>>> **not** True

False

>>> **not** False

True

>>> **not** 1 > 2

True

布尔值经常用在条件判断中，比如：

**if** age >= 18:

**print**('adult')

**else**:

**print**('teenager')

#### **空值**

空值是Python里一个特殊的值，用None表示。None不能理解为0，因为0是有意义的，而None是一个特殊的空值。

此外，Python还提供了列表、字典等多种数据类型，还允许创建自定义数据类型，我们后面会继续讲到。

## 变量

变量的概念基本上和初中代数的方程变量是一致的，只是在计算机程序中，变量不仅可以是数字，还可以是任意数据类型。

变量在程序中就是用一个变量名表示了，变量名必须是大小写英文、数字和\_的组合，且不能用数字开头，比如：

a = 1

变量a是一个整数。

t\_007 = 'T007'

变量t\_007是一个字符串。

Answer = **True**

变量Answer是一个布尔值True。

在Python中，等号=是赋值语句，可以把任意数据类型赋值给变量，同一个变量可以反复赋值，而且可以是不同类型的变量，例如：

a = 123 *# a是整数*

**print**(a)

a = 'ABC' *# a变为字符串*

**print**(a)

这种变量本身类型不固定的语言称之为动态语言，与之对应的是静态语言。静态语言在定义变量时必须指定变量类型，如果赋值的时候类型不匹配，就会报错。例如Java是静态语言，赋值语句如下（// 表示注释）：

**int** a = 123; *// a是整数类型变量*

a = "ABC"; *// 错误：不能把字符串赋给整型变量*

和静态语言相比，动态语言更灵活，就是这个原因。

请不要把赋值语句的等号等同于数学的等号。比如下面的代码：

x = 10

x = x + 2

如果从数学上理解x = x + 2那无论如何是不成立的，在程序中，赋值语句先计算右侧的表达式x + 2，得到结果12，再赋给变量x。由于x之前的值是10，重新赋值后，x的值变成12。

最后，理解变量在计算机内存中的表示也非常重要。当我们写：

a = 'ABC'

时，Python解释器干了两件事情：

1. 在内存中创建了一个'ABC'的字符串；
2. 在内存中创建了一个名为a的变量，并把它指向'ABC'。

也可以把一个变量a赋值给另一个变量b，这个操作实际上是把变量b指向变量a所指向的数据，例如下面的代码：

a = 'ABC'

b = a

a = 'XYZ'

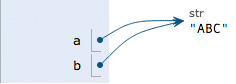
**print**(b)

最后一行打印出变量b的内容到底是'ABC'呢还是'XYZ'？如果从数学意义上理解，就会错误地得出b和a相同，也应该是'XYZ'，但实际上b的值是'ABC'，让我们一行一行地执行代码，就可以看到到底发生了什么事：

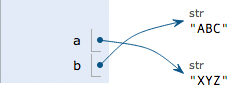
执行a = 'ABC'，解释器创建了字符串'ABC'和变量a，并把a指向'ABC'：

py-var-code-1

执行b = a，解释器创建了变量b，并把b指向a指向的字符串'ABC'：



执行a = 'XYZ'，解释器创建了字符串'XYZ'，并把a的指向改为'XYZ'，但b并没有更改：



所以，最后打印变量b的结果自然是'ABC'了。

## 常量

所谓常量就是不能变的变量，比如常用的数学常数π就是一个常量。在Python中，通常用全部大写的变量名表示常量：

PI = 3.14159265359

但事实上PI仍然是一个变量，Python根本没有任何机制保证PI不会被改变，所以，用全部大写的变量名表示常量只是一个习惯上的用法，如果你一定要改变变量PI的值，也没人能拦住你。

最后解释一下整数的除法为什么也是精确的。在Python中，有两种除法，一种除法是/：

>>> 10 / 3

3.3333333333333335

/除法计算结果是浮点数，即使是两个整数恰好整除，结果也是浮点数：

>>> 9 / 3

3.0

还有一种除法是//，称为地板除，两个整数的除法仍然是整数：

>>> 10 *// 3*

3

你没有看错，整数的地板除//永远是整数，即使除不尽。要做精确的除法，使用/就可以。

因为//除法只取结果的整数部分，所以Python还提供一个余数运算，可以得到两个整数相除的余数：

>>> 10 % 3

1

无论整数做//除法还是取余数，结果永远是整数，所以，整数运算结果永远是精确的。

练习

请打印出以下变量的值：

窗体顶端

n = 123

f = 456.789

s1 = 'Hello, world'

s2 = 'Hello, \'Adam\''

s3 = r'Hello, "Bart"'

s4 = r'''Hello,

Lisa!'''

 Run

窗体底端

**小结**

Python支持多种数据类型，在计算机内部，可以把任何数据都看成一个“对象”，而变量就是在程序中用来指向这些数据对象的，对变量赋值就是把数据和变量给关联起来。

注意：Python的整数没有大小限制，而某些语言的整数根据其存储长度是有大小限制的，例如Java对32位整数的范围限制在-2147483648-2147483647。

Python的浮点数也没有大小限制，但是超出一定范围就直接表示为inf（无限大）。

## 字符串与编码

我们已经讲过了，字符串也是一种数据类型，但是，字符串比较特殊的是还有一个编码问题。

因为计算机只能处理数字，如果要处理文本，就必须先把文本转换为数字才能处理。最早的计算机在设计时采用8个比特（bit）作为一个字节（byte），所以，一个字节能表示的最大的整数就是255（二进制11111111=十进制255），如果要表示更大的整数，就必须用更多的字节。比如两个字节可以表示的最大整数是65535，4个字节可以表示的最大整数是4294967295。

由于计算机是美国人发明的，因此，最早只有127个字符被编码到计算机里，也就是大小写英文字母、数字和一些符号，这个编码表被称为ASCII编码，比如大写字母A的编码是65，小写字母z的编码是122。

但是要处理中文显然一个字节是不够的，至少需要两个字节，而且还不能和ASCII编码冲突，所以，中国制定了GB2312编码，用来把中文编进去。

你可以想得到的是，全世界有上百种语言，日本把日文编到Shift\_JIS里，韩国把韩文编到Euc-kr里，各国有各国的标准，就会不可避免地出现冲突，结果就是，在多语言混合的文本中，显示出来会有乱码。



因此，Unicode应运而生。Unicode把所有语言都统一到一套编码里，这样就不会再有乱码问题了。

Unicode标准也在不断发展，但最常用的是用两个字节表示一个字符（如果要用到非常偏僻的字符，就需要4个字节）。现代操作系统和大多数编程语言都直接支持Unicode。

现在，捋一捋ASCII编码和Unicode编码的区别：ASCII编码是1个字节，而Unicode编码通常是2个字节。

字母A用ASCII编码是十进制的65，二进制的01000001；

字符0用ASCII编码是十进制的48，二进制的00110000，注意字符'0'和整数0是不同的；

汉字中已经超出了ASCII编码的范围，用Unicode编码是十进制的20013，二进制的01001110 00101101。

你可以猜测，如果把ASCII编码的A用Unicode编码，只需要在前面补0就可以，因此，A的Unicode编码是00000000 01000001。

新的问题又出现了：如果统一成Unicode编码，乱码问题从此消失了。但是，如果你写的文本基本上全部是英文的话，用Unicode编码比ASCII编码需要多一倍的存储空间，在存储和传输上就十分不划算。

所以，本着节约的精神，又出现了把Unicode编码转化为“可变长编码”的UTF-8编码。UTF-8编码把一个Unicode字符根据不同的数字大小编码成1-6个字节，常用的英文字母被编码成1个字节，汉字通常是3个字节，只有很生僻的字符才会被编码成4-6个字节。如果你要传输的文本包含大量英文字符，用UTF-8编码就能节省空间：

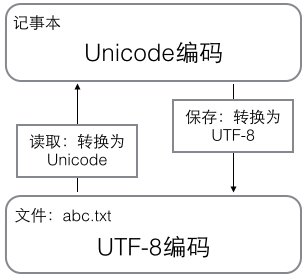
|  |  |  |  |
| --- | --- | --- | --- |
| **字符** | **ASCII** | **Unicode** | **UTF-8** |
| A | 01000001 | 00000000 01000001 | 01000001 |
| 中 | x | 01001110 00101101 | 11100100 10111000 10101101 |

从上面的表格还可以发现，UTF-8编码有一个额外的好处，就是ASCII编码实际上可以被看成是UTF-8编码的一部分，所以，大量只支持ASCII编码的历史遗留软件可以在UTF-8编码下继续工作。

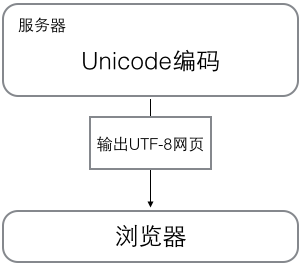
搞清楚了ASCII、Unicode和UTF-8的关系，我们就可以总结一下现在计算机系统通用的字符编码工作方式：

在计算机内存中，统一使用Unicode编码，当需要保存到硬盘或者需要传输的时候，就转换为UTF-8编码。

用记事本编辑的时候，从文件读取的UTF-8字符被转换为Unicode字符到内存里，编辑完成后，保存的时候再把Unicode转换为UTF-8保存到文件：



浏览网页的时候，服务器会把动态生成的Unicode内容转换为UTF-8再传输到浏览器：



所以你看到很多网页的源码上会有类似<meta charset="UTF-8" />的信息，表示该网页正是用的UTF-8编码。

## Python的字符串

搞清楚了令人头疼的字符编码问题后，我们再来研究Python的字符串。

在最新的Python 3版本中，字符串是以Unicode编码的，也就是说，Python的字符串支持多语言，例如：

>>> print('包含中文的str')

包含中文的str

对于单个字符的编码，Python提供了ord()函数获取字符的整数表示，chr()函数把编码转换为对应的字符：

>>> ord('A')

65

>>> ord('中')

20013

>>> chr(66)

'B'

>>> chr(25991)

'文'

如果知道字符的整数编码，还可以用十六进制这么写str：

>>> '\u4e2d\u6587'

'中文'

两种写法完全是等价的。

由于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通过encode()方法可以编码为指定的bytes，例如：

>>> 'ABC'.encode('ascii')

b'ABC'

>>> '中文'.encode('utf-8')

b'\xe4\xb8\xad\xe6\x96\x87'

>>> '中文'.encode('ascii')

Traceback (most recent call last):

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

UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-1: ordinal not in range(128)

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。

反过来，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，就需要用decode()方法：

>>> b'ABC'.decode('ascii')

'ABC'

>>> b'\xe4\xb8\xad\xe6\x96\x87'.decode('utf-8')

'中文'

要计算str包含多少个字符，可以用len()函数：

>>> len('ABC')

3

>>> len('中文')

2

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：

>>> len(b'ABC')

3

>>> len(b'\xe4\xb8\xad\xe6\x96\x87')

6

>>> len('中文'.encode('utf-8'))

6

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1个字节。

在操作字符串时，我们经常遇到str和bytes的互相转换。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：

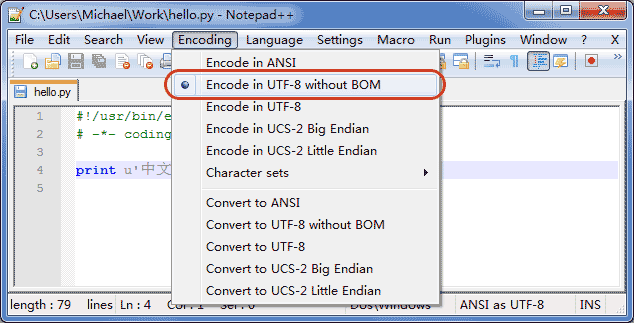
*#!/usr/bin/env python3*

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

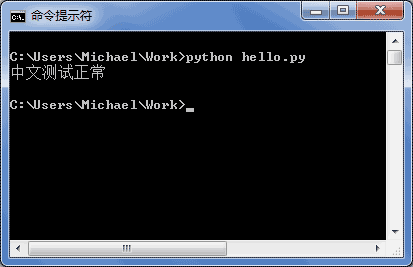
第一行注释是为了告诉Linux/OS X系统，这是一个Python可执行程序，Windows系统会忽略这个注释；

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：

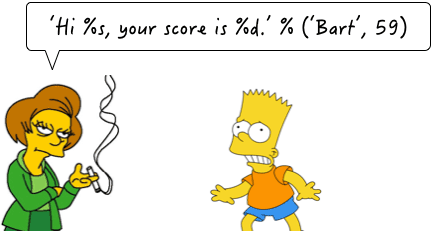


如果.py文件本身使用UTF-8编码，并且也申明了# -\*- coding: utf-8 -\*-，打开命令提示符测试就可以正常显示中文：



## 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，举例如下：

>>> 'Hello, %s' % 'world'

'Hello, world'

>>> 'Hi, %s, you have $%d.' % ('Michael', 1000000)

'Hi, Michael, you have $1000000.'

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%?，括号可以省略。

常见的占位符有：

|  |  |
| --- | --- |
| %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：

>>> '%2d-%02d' % (3, 1)

' 3-01'

>>> '%.2f' % 3.1415926

'3.14'

如果你不太确定应该用什么，%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 %'

练习

小明的成绩从去年的72分提升到了今年的85分，请计算小明成绩提升的百分点，并用字符串格式化显示出'xx.x%'，只保留小数点后1位：

窗体顶端

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

s1 = 72

s2 = 85



答案1：

s1=72

s2=85

r=(s2-s1)/s1\*100

print('%s scroe is up :%s%%'%('xiaoming','%.1f'%r))

答案2：

s1=72

s2=85

print ('%s scroe is up :%s%%'%('xiaoming','%.1f'%float((s2-s1)/s1\*100)))

窗体底端

**小结**

Python 3的字符串使用Unicode，直接支持多语言。

str和bytes互相转换时，需要指定编码。最常用的编码是UTF-8。Python当然也支持其他编码方式，比如把Unicode编码成GB2312：

>>> '中文'.encode('gb2312')

b'\xd6\xd0\xce\xc4'

但这种方式纯属自找麻烦，如果没有特殊业务要求，请牢记仅使用UTF-8编码。

格式化字符串的时候，可以用Python的交互式命令行测试，方便快捷。

## 使用list和tuple

### **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其中的元素。

比如，列出班里所有同学的名字，就可以用一个list表示：

>>> classmates = ['Michael', 'Bob', 'Tracy']

>>> 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：

>>> len(classmates)

3

用索引来访问list中每一个位置的元素，记得索引是从0开始的：

>>> classmates[0]

'Michael'

>>> classmates[1]

'Bob'

>>> classmates[2]

'Tracy'

>>> classmates[3]

Traceback (most recent call last):

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

IndexError: list index out of range

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(classmates) - 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[-1]

'Tracy'

以此类推，可以获取倒数第2个、倒数第3个：

>>> 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，比如：

>>> L = ['Apple', 123, True]

list元素也可以是另一个list，比如：

>>> 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，如果拆开写就更容易理解了：

>>> p = ['asp', 'php']

>>> s = ['python', 'java', p, 'scheme']

要拿到'php'可以写p[1]或者s[2][1]，因此s可以看成是一个二维数组，类似的还有三维、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：

>>> L = []

>>> len(L)

0

### **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。

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：当你定义一个tuple时，在定义的时候，tuple的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，如果你这么定义：

>>> t = (1)

>>> t

1

定义的不是tuple，是1这个数！这是因为括号()既可以表示tuple，又可以表示数学公式中的小括号，这就产生了歧义，因此，Python规定，这种情况下，按小括号进行计算，计算结果自然是1。

所以，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3个元素，分别是'a'，'b'和一个list。不是说tuple一旦定义后就不可变了吗？怎么后来又变了？

别急，我们先看看定义的时候tuple包含的3个元素：



当我们把list的元素'A'和'B'修改为'X'和'Y'后，tuple变为：



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并没有改成别的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。

练习

请用索引取出下面list的指定元素：

窗体顶端

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

L = [

['Apple', 'Google', 'Microsoft'],

['Java', 'Python', 'Ruby', 'PHP'],

['Adam', 'Bart', 'Lisa']

]

 Run

窗体底端

**小结**

list和tuple是Python内置的有序集合，一个可变，一个不可变。根据需要来选择使用它们。

## 条件判断

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。

比如，输入用户年龄，根据年龄打印不同的内容，在Python程序中，用if语句实现：

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，就把缩进的两行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，把该判断对应的语句执行后，就忽略掉剩下的elif和else，所以，请测试并解释为什么下面的程序打印的是teenager：

age = 20

**if** age >= 6:

print('teenager')

**elif** age >= 18:

print('adult')

**else**:

print('kid')

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：

Traceback (most recent call last):

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

TypeError: unorderable types: str() > int()

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')

再次运行，就可以得到正确地结果。但是，如果输入abc呢？又会得到一个错误信息：

Traceback (most recent **call** **last**):

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

ValueError: invalid literal **for** **int**() **with** base 10: 'abc'

原来int()函数发现一个字符串并不是合法的数字时就会报错，程序就退出了。

如何检查并捕获程序运行期的错误呢？后面的错误和调试会讲到。

练习

小明身高1.75，体重80.5kg。请根据BMI公式（体重除以身高的平方）帮小明计算他的BMI指数，并根据BMI指数：

* 低于18.5：过轻
* 18.5-25：正常
* 25-28：过重
* 28-32：肥胖
* 高于32：严重肥胖

用if-elif判断并打印结果：

窗体顶端

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

height = 1.75

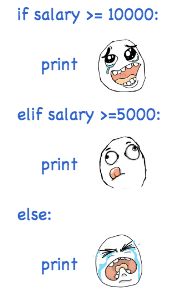
weight = 80.5

 Run

窗体底端

小结

条件判断可以让计算机自己做选择，Python的if...elif...else很灵活。



## 循环

要计算1+2+3，我们可以直接写表达式：

>>> 1 + 2 + 3

6

要计算1+2+3+...+10，勉强也能写出来。

但是，要计算1+2+3+...+10000，直接写表达式就不可能了。

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：

Michael

Bob

Tracy

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

再比如我们想计算1-10的整数之和，可以用一个sum变量做累加：

sum = 0

**for** x **in** [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:

sum = sum + x

print(sum)

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，可以生成一个整数序列，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]

range(101)就可以生成0-100的整数序列，计算如下：

sum = 0

**for** x **in** range(101):

sum = sum + x

print(sum)

请自行运行上述代码，看看结果是不是当年高斯同学心算出的5050。

第二种循环是while循环，只要条件满足，就不断循环，条件不满足时退出循环。比如我们要计算100以内所有奇数之和，可以用while循环实现：

sum = 0

n = 99

**while** n > 0:

sum = sum + n

n = n - 2

**print**(sum)

在循环内部变量n不断自减，直到变为-1时，不再满足while条件，循环退出。

练习

请利用循环依次对list中的每个名字打印出Hello, xxx!：

窗体顶端

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

L = ['Bart', 'Lisa', 'Adam']

 Run

窗体底端

### **break**

在循环中，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：

n = 1

while n <= 100:

print(n)

n = n + 1

print('END')

上面的代码可以打印出1~100。

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。

n = 0

while n < 10:

n = n + 1

print(n)

上面的程序可以打印出1～10。但是，如果我们想只打印奇数，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)

执行上面的代码可以看到，打印的不再是1～10，而是1，3，5，7，9。

可见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

循环是让计算机做重复任务的有效的方法。

break语句可以在循环过程中直接退出循环，而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。大多数循环并不需要用到break和continue语句，上面的两个例子，都可以通过改写循环条件或者修改循环逻辑，去掉break和continue语句。

有些时候，如果代码写得有问题，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。

请试写一个死循环程序。

## 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，就先要在names中找到对应的位置，再从scores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，查找速度都不会变慢。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。

dict就是第二种实现方式，给定一个名字，比如'Michael'，dict在内部就可以直接计算出Michael对应的存放成绩的“页码”，也就是95这个数字存放的内存地址，直接取出来，所以速度非常快。

你可以猜到，这种key-value存储方式，在放进去的时候，必须根据key算出value的存放位置，这样，取的时候才能根据key直接拿到value。

把数据放入dict的方法，除了初始化时指定外，还可以通过key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key不存在，dict就会报错：

>>> d['Thomas']

Traceback (most recent call last):

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

KeyError: 'Thomas'

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的交互式命令行不显示结果。

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；
2. 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**。

这是因为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key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，可以放心地作为key。而list是可变的，就不能作为key：

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

>>> d[key] = 'a list'

Traceback (most recent call last):

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

TypeError: unhashable type: 'list'

set

set和dict类似，也是一组key的集合，但不存储value。由于key不能重复，所以，在set中，没有重复的key。

要创建一个set，需要提供一个list作为输入集合：

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

>>> s

{1, 2, 3}

注意，传入的参数[1, 2, 3]是一个list，而显示的{1, 2, 3}只是告诉你这个set内部有1，2，3这3个元素，显示的顺序也不表示set是有序的。。

重复元素在set中自动被过滤：

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

>>> s

{1, 2, 3}

通过add(key)方法可以添加元素到set中，可以重复添加，但不会有效果：

>>> s.add(4)

>>> s

{1, 2, 3, 4}

>>> s.add(4)

>>> s

{1, 2, 3, 4}

通过remove(key)方法可以删除元素：

>>> s.remove(4)

>>> s

{1, 2, 3}

set可以看成数学意义上的无序和无重复元素的集合，因此，两个set可以做数学意义上的交集、并集等操作：

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

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

>>> s1 & s2

{2, 3}

>>> s1 | s2

{1, 2, 3, 4}

set和dict的唯一区别仅在于没有存储对应的value，但是，set的原理和dict一样，所以，同样不可以放入可变对象，因为无法判断两个可变对象是否相等，也就无法保证set内部“不会有重复元素”。试试把list放入set，看看是否会报错。

再议不可变对象

上面我们讲了，str是不变对象，而list是可变对象。

对于可变对象，比如list，对list进行操作，list内部的内容是会变化的，比如：

>>> a = ['c', 'b', 'a']

>>> a.sort()

>>> a

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

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'

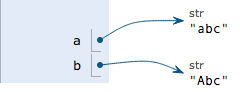
>>> a

'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'：

a-to-str

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。

小结

使用key-value存储结构的dict在Python中非常有用，选择不可变对象作为key很重要，最常用的key是字符串。

tuple虽然是不变对象，但试试把(1, 2, 3)和(1, [2, 3])放入dict或set中，并解释结果。

**修改了，看到没**