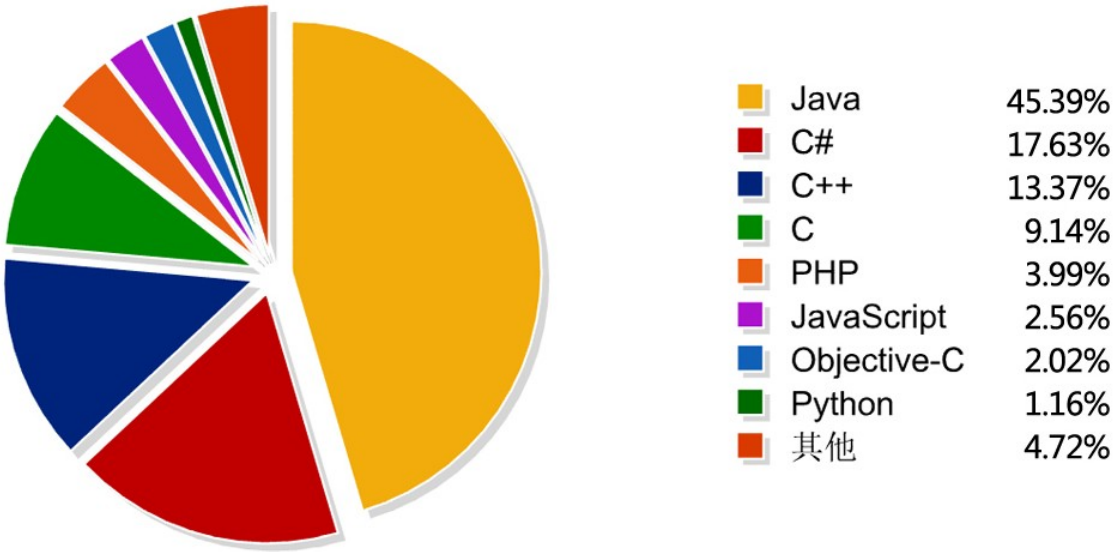
# 数据结构与算法

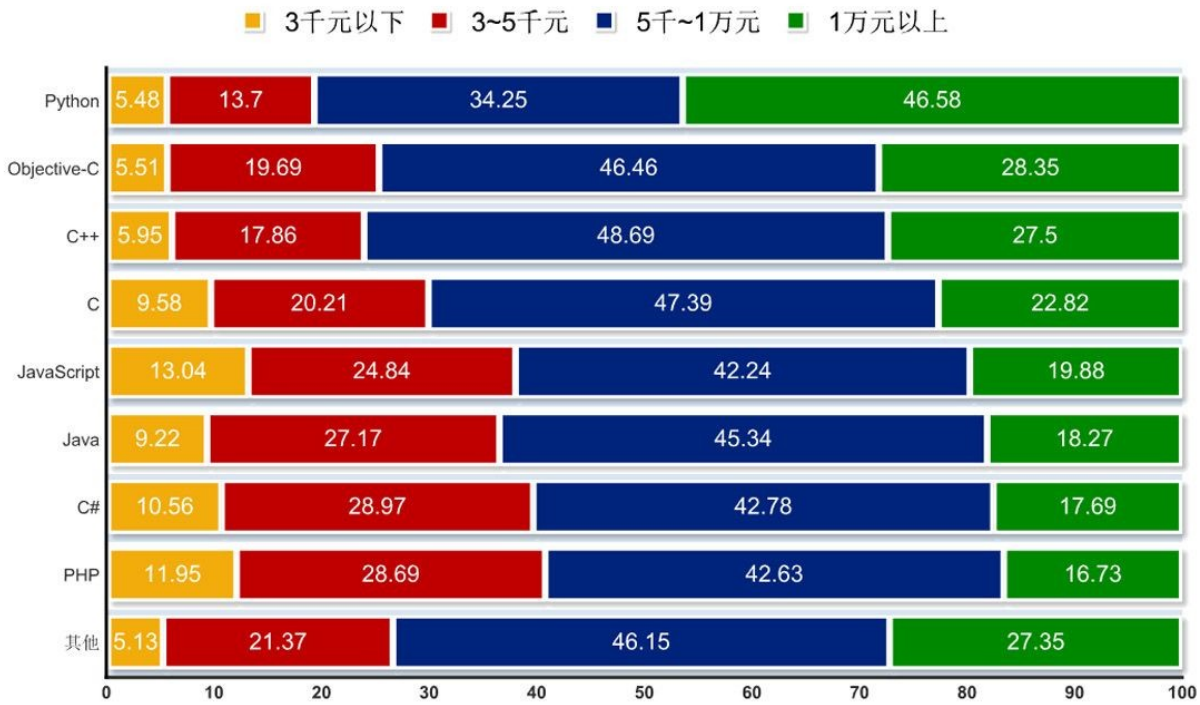
# -Python实现

## 张晓平

参考资料：

problem solving with algorithms and data structure using python 中文版 (https://facert.gitbooks.io/python-data-structure-cn/)






需要的工具：

- 安装 Python
- 安装 jupyter notebook

Ubuntu 16.04下：

1 安装python和python-pip

```
* sudo apt-get install python python3 python-pip python3-pip
* sudo pip install --upgrade pip  #更新pip
* sudo pip3 install --upgrade pip
```

2 安装jupyter-notebook

```
* sudo pip install jupyter
* sudo pip3 install jupyter
```

3 配置可以同时使用python2和python3内核

```
* sudo ipython kernel install--user
* sudo python3 -m ipykernel install--user
* sudo pip2 install -U ipykernel
* sudo python2 -m ipykernel install--user
```

# Python基础

参考资料：

Python教程 by 廖雪峰 (https://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000)

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，最大的区别是：

- 自然语言在不同的语境下有不同的理解；
- 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：

In [1]:
```
# print absolute value of an integer:
a = 100
if a >= 0:
    print(a)
else:
    print(-a)
```

```
100
```

- 注释
  - 以#开头
  - 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。

- 缩进之利
  - 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*。
  - 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。

- 缩进之弊
  - “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。
  - IDE很难像格式化Java代码那样格式化Python代码。

- 区分大小写

# 小结

- 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

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
- 浮点数
- 字符串
- 布尔值
- 空值

## 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，如：

1，100，-8080，0，...

计算机使用二进制，但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，如：

0xff00，0xa5b4c3d2，...

## 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，...

- 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。 如：
  - $1.23 \times 10^9$ 就是1.23e9，12.3e8
  - 0.000012可以写成1.2e-5

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，如:

'abc'，"xyz"，...

注意：

- 单引号或双引号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，那就可以用双引号括起来。

  比如"I'm OK"包含六个字符： I，'，m，空格，O，K。

注意

- 如果字符串内部既包含单引号又包含双引号怎么办？

  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 : 换行，
- \t : 制表符，
- \ : 表示字符\

以下代码用print()打印字符串看看：

In [3]:
```
print('I\'m ok.')
print('I\'m learning\nPython.')
print('\\\n\\')
```

```
I'm ok.
I'm learning
Python.
\
\
```

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。例如：

In [4]:
```
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。例如：

In [3]:
```
print('''line1
line2
line3''')
```

```
line1
line2
line3
```

## 布尔值

布尔值和布尔代数的表示完全一致，一个布尔值只有两种选择：

- True
- False

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：

In [19]:
```
print True
print False
print 3 > 2
print 3 > 5
```

```
True
False
True
False
```

布尔值有三种运算：

- and
- or
- not

and: 逻辑与运算。

只有所有都为True，and运算结果才是True。

In [23]:
```
print True and True
print True and False
print False and True
print False and False
```

```
True
False
False
False
```

or：逻辑或运算。

只要其中有一个为True，or运算结果就是True。

In [24]:
```
print True or True
print True or False
print False or True
print False or False
```

```
True
True
True
False
```

not：逻辑非运算，是一个单目运算符。

把True变成False，False变成True：

In [25]:
```
print not True
print not False
print not 1 > 2
```

```
False
True
True
```

布尔值经常用在条件判断中，比如：

In [27]:
```
age = 12
if age >= 18:
    print('adult')
else:
    print('teenager')
```

```
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：

In [28]:
```
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：

In [29]:
```
a = 123 # a是整数
print(a)
a = 'ABC' # a变为字符串
print(a)
```

```
123
ABC
```

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 C 是静态语言，赋值语句如下（// 表示注释）：

In [ ]:
```
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：

In [31]:
```
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：

- 在内存中创建了一个 'ABC' 的字符串；
- 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：

In [33]:
```
a = 'ABC'
b = a
a = 'XYZ'
print(b)
```

```
ABC
```

# 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：

In [34]:
```
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。

因此，用全部大写的变量名表示常量只是一个习惯上的用法。如果你非要改变PI的值，也没人能拦住你。

### 整数的除法

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。

一种是 /。

在 Python3 中，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。

In [40]:
```
print(10 / 3)
print(9 / 3)
```

```
3.3333333333333335
3.0
```

另一种是 //，称为地板除，两个整数的除法仍然是整数。

除法 // 只取结果的整数部分。

In [7]:
```
print(10 // 3)
print(9 // 3)
print(10.0 // 3.0)
print(9.0 // 3.0)
```

```
3
3
3.0
3.0
```

Python 还提供一个余数运算，可以得到两个整数相除的余数：

In [9]:
```
print(10 % 3)
```

```
1
```

## 练习

请打印出以下变量的值：

In [42]:
```
n = 123
f = 456.789
s1 = 'Hello, world'
s2 = 'Hello, \'Adam\''
s3 = r'Hello, "Bart"'
s4 = r'''Hello,
Lisa!'''
```

## 小结

Python支持多种数据类型。

在计算机内部，

- 任何数据都可看成一个“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

字符串是一种数据类型，但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，因$(11111111)_2 = 255$。
- 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，
  - 四个字节可以表示的最大整数是 4294967295。

### ASCII、Unicode 和 UTF-8 编码

#### ASCII 编码

由于计算机是美国人发明的，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。

- 大写字母 A 的编码是 65，小写字母 z 的编码是 122。

#### Unicode编码

- 处理中文的话一个字节显然不够，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。
- 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。

- 'A' 用 ASCII编码是$65$，亦即$(01000001)_2$；
- '0' 用 ASCII编码是$48$，亦即$(00110000)_2$，注意 '0'和 0 是不同的；
- '中'已经超出了 ASCII编码的范围，用 Unicode编码是$20013$，即$(01001110\ \ 00101101)_2$。

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$(00000000\ \ 01000001)_2$。

**UTF-8**编码

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' | | 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。

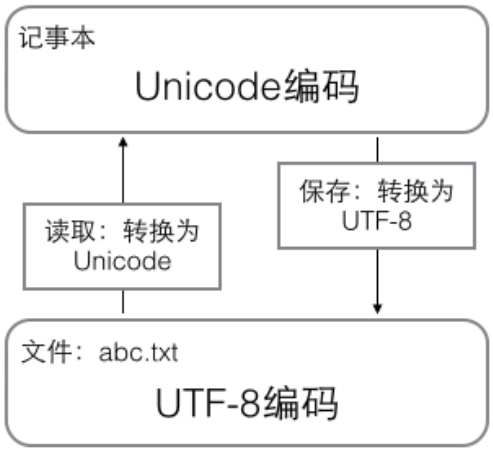
所以，大量只支持 ASCII编码的历史遗留软件可以在 UTF-8编码下继续工作。

字符编码工作方式

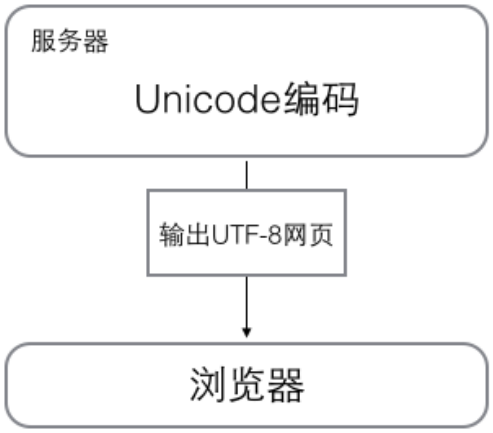
搞清楚了 ASCII、Unicode和 UTF-8的关系，我们就可以总结一下现在计算机系统通用的字符编码工作方式：

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Python3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：

In [15]:
```
print('包含中文的str')
```

```
包含中文的str
```

对于单个字符的编码，Python提供了两个函数来实现字符与编码的互换：

- ord(): 获取字符的整数表示；
- chr()：把编码转换为对应的字符。

In [14]:
```
print(ord('A'))
print(ord('中'))

print(chr(66))
print(chr(25991))
```

```
65
20013
B
文
```

如果知道字符的整数编码，还可以用十六进制这么写str：

In [13]:
```
'\u4e2d\u6587'
```

Out[13]: '中文'

## str 和 bytes 的互相转换

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

(1) 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：

In [65]:
```
x = b'ABC'
print x
```

```
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：

In [12]:
```
print('ABC'.encode('ascii'))
print('中文'.encode('utf-8'))
print('中文'.encode('ascii'))
```

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

---------------------------------------------------------------------------
UnicodeEncodeError                        Traceback (most recent call last)
<ipython-input-12-065169672c83> in <module>()
      1 print('ABC'.encode('ascii'))
      2 print('中文'.encode('utf-8'))
----> 3 print('中文'.encode('ascii'))

UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-1: or
dinal 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。此时，在bytes中无法显示为ASCII字符的字节，用\x##显示。
- 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。

(2) 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：

In [10]:
```
print(b'ABC'.decode('ascii'))
print(b'\xe4\xb8\xad\xe6\x96\x87'.decode('utf-8'))
```

```
ABC
中文
```

## len()

要计算 str 包含多少个字符，可以用 len()：

In [9]:
```
print(len('ABC'))
print(len('中文'))
```

```
3
2
```

len()计算str的字符数。如果换成bytes，len()函数就计算字节数。

In [4]:
```
print(len(b'ABC'))
print(len(b'\xe4\xb8\xad\xe6\x96\x87'))
print(len('中文'.encode('utf-8')))
```

```
3
6
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。

## 坚持使用 **UTF-8** 编码

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。

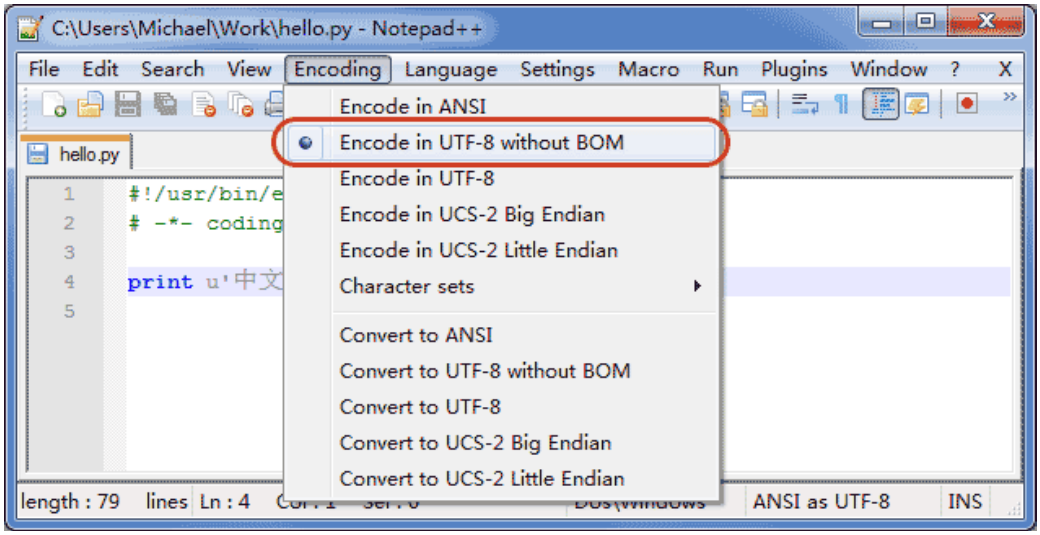
由于Python源代码也是一个文本文件，如果源代码中包含中文，在保存源代码时，就需要务必指定保存为UTF-8编码。

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：

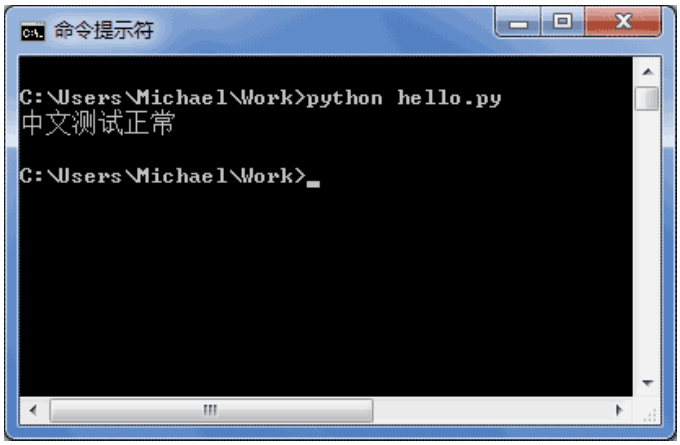
In [73]:
```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
```

- 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

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。于是，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：

In [53]:
```
print('Hello, %s' % 'world')
print('Hi, %s, you have $%d.' % ('Michael', 1000000))
```

```
Hello, world
Hi, Michael, you have $1000000.
```

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 |

整数和浮点数的格式化还可以指定是否补0、是否表明整数与小数的位数：

In [79]:
```
print('%2d-%02d' % (3, 1))
print('%.2f' % 3.1415926)
```

```
 3-01
3.14
```

若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：

In [78]:
```
print('Age: %s. Gender: %s' % (25, True))
```

```
Age: 25. Gender: True
```

若想输出 %，请用 %%：

In [80]:
```
print('growth rate: %d %%' % 7)
```

```
growth rate: 7 %
```

## 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：

In [54]:
```
print('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。

# 容器（container）

- 列表
- 元组
- 字典
- 集合

# 列表

list是Python内置的一种数据类型，它是一种有序的集合，可以随时添加和删除其中的元素。

用方括号[]创建列表。 比如，列出班里所有同学的名字，可以这样做：

In [38]:
```
classmates = ['Michael', 'Bob', 'Tracy']
classmates
```

Out[38]:
```
['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：

In [39]:
```
len(classmates)
```

Out[39]:
```
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：

In [40]:
```
print(classmates[0])
print(classmates[1])
print(classmates[2])
print(classmates[3])
```

```
Michael
Bob
Tracy

---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-40-29e5e303dfc1> in <module>()
      2 print(classmates[1])
      3 print(classmates[2])
----> 4 print(classmates[3])

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：

In [41]:
```
print(classmates[-1])
print(classmates[-2])
print(classmates[-3])
print(classmates[-4])
```

```
Tracy
Bob
Michael

---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-41-ae30820961d0> in <module>()
      2 print(classmates[-2])
      3 print(classmates[-3])
----> 4 print(classmates[-4])

IndexError: list index out of range
```

list是一个可变的有序表，有多种操作方法。

可以往list中追加元素到末尾：

In [45]:
```
classmates.append('Adam')
print(classmates)
```

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

可以把元素插入到指定的位置，比如索引号为1的位置：

In [43]:
```
classmates.insert(1, 'Jack')
print(classmates)
```

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

要删除list末尾的元素，用pop()方法：

In [44]:
```
classmates.pop()
print(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：

In [46]:
```
classmates.pop(1)
print(classmates)
```

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

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：

In [47]:
```
classmates[1] = 'Sarah'
print(classmates)
```

```
['Michael', 'Sarah', 'Tracy', 'Adam']
```

list里面的元素的数据类型也可以不同，比如：

In [48]:
```
L = ['Apple', 123, True]
```

list元素也可以是另一个list，比如：

In [49]:
```
s = ['python', 'java', ['asp', 'php'], 'scheme']
```

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：

In [14]:
```
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。

In [51]:
```
L = []
len(L)
```

Out[51]: 0

# 元组(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：

In [52]:
```
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：

In [53]:
```
t = (1, 2)
t
```

Out[53]: (1, 2)

如果要定义一个空的tuple，可以写成()：

In [54]:
```
t = ()
t
```

Out[54]: ()

但是，要定义一个只有1个元素的tuple，如果你这么定义：

In [56]:
```
t = (1)
t
```

Out[56]: 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。

只有1个元素的tuple定义时必须加一个逗号,，来消除歧义：

In [59]:
```
t = (1,)
t
```

Out[59]: (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。

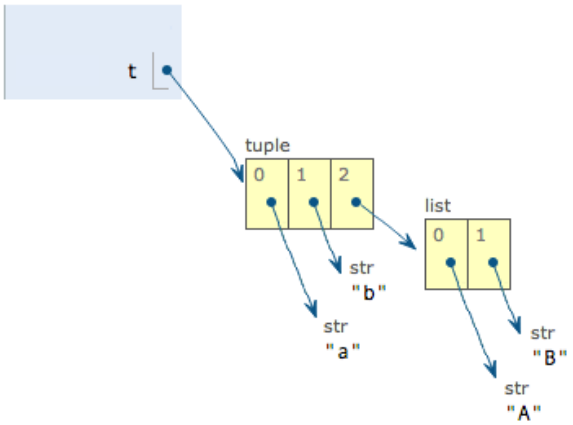
最后来看一个“可变的”tuple：

In [60]:
```
t = ('a', 'b', ['A', 'B'])
t[2][0] = 'X'
t[2][1] = 'Y'
t
```

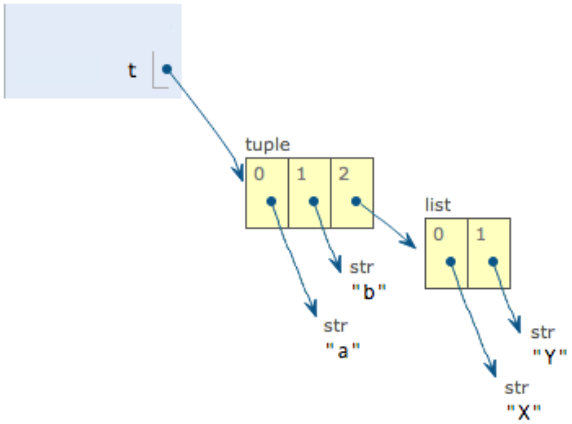
Out[60]: ('a', 'b', ['X', 'Y'])

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：

In [ ]:
```
L = [
    ['Apple', 'Google', 'Microsoft'],
    ['Java', 'Python', 'Ruby', 'PHP'],
    ['Adam', 'Bart', 'Lisa']
]

# 打印Apple:
print(?)
# 打印Python:
print(?)
# 打印Lisa:
print(?)
```

## 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。

# 字典

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：

In [17]:
```
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。

In [23]:
```
d = {'Michael': 95, 'Bob': 75, 'Tracy': 85}
d['Michael']
```

Out[23]:
```
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，

- 第一种方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：

In [24]:
```
d['Adam'] = 67
d
```

Out[24]:
```
{'Adam': 67, 'Bob': 75, 'Michael': 95, 'Tracy': 85}
```

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：

In [25]:
```
d['Jack'] = 90
print(d['Jack'])

d['Jack'] = 88
print(d['Jack'])
```

```
90
88
```

如果key不存在，dict就会报错：

In [26]:
```
d['Thomas']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-26-d933a34735ff> in <module>()
----> 1 d['Thomas']

KeyError: 'Thomas'
```

要避免key不存在的错误，有两种办法:

(1) 通过 in 判断key是否存在：

In [27]:
```
'Thomas' in d
```

Out[27]:
```
False
```

(2) 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：

In [28]:
```
print(d.get('Thomas'))
print(d.get('Thomas', -1))
```

```
None
-1
```

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

In [82]:
```
d.pop('Bob')
d
```

Out[82]:
```
{'Adam': 67, 'Jack': 88, '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，

- 整数、字符串、元组等都是不可变的，因此，可以放心地作为key。
- 而list是可变的，就不能作为key：

In [31]:
```
key = (1, 2, 3)
d[key] = 'alist'

key = [1, 2, 3]
d[key] = 'alist'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-31-60912712423b> in <module>()
      3
      4 key = [1, 2, 3]
----> 5 d[key] = 'alist'

TypeError: unhashable type: 'list'
```

# 集合(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。

创建一个set，有以下两种方法：

(1) 可使用花括号{}：

In [92]:
```
s = {1, 2, 3}
print(s)
print(type(s))
```

```
{1, 2, 3}
<class 'set'>
```

（2）也可提供一个list作为输入集合。

In [32]:
```
s = set([1, 2, 3])
print(s)
print(type(s))
```

```
{1, 2, 3}
<class 'set'>
```

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：

In [94]:
```
s = {1, 1, 2, 2, 3, 3}
print(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：

In [97]:
```
s.add(4)
print(s)

s.add(4)
print(s)
```

```
{1, 2, 3, 4}
{1, 2, 3, 4}
```

通过remove(key)方法可以删除元素：

In [98]:
```
s.remove(4)
print(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：

In [100]:
```
s1 = {1, 2, 3}
s2 = {2, 3, 4}
print(s1 & s2)  # 交集
print(s1 | s2)  # 并集
```

```
{2, 3}
{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

我们已经知道，str是不变对象，而list是可变对象。

对于可变对象，比如list，对其进行操作，其内部内容是会变化的。

In [33]:
```
a = ['c', 'b', 'a']
a.sort()
print(a)
```

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

而对于不可变对象，比如str，对其进行操作呢？

In [103]:
```
a = 'abc'
a.replace('a', 'A')
print(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：

In [104]:
```
a = 'abc'
b = a.replace('a', 'A')

print(b)
print(a)
```

```
Abc
abc
```

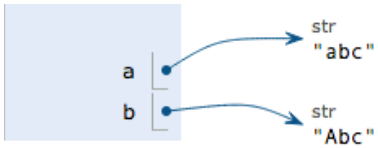
要始终牢记的是，a是变量，而'abc'才是字符串对象！

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，虽然方法replace作用在字符串对象'abc'上，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。

# 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，再根据年龄打印不同的内容。

可用if语句实现：

In [107]:
```
age = 20
if age >= 18:
    print('your age is', age)
    print('adult')
```

```
your age is 20
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。如果if判断是False，不要执行if的内容，去把else执行了：

In [109]:
```
age = 3
if age >= 18:
    print('your age is', age)
    print('adult')
else:
    print('your age is', age)
    print('teenager')
```

```
your age is 3
teenager
```

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：

In [110]:
```
age = 3
if age >= 18:
    print('adult')
elif age >= 6:
    print('teenager')
else:
    print('kid')
```

```
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：

In [ ]:
```
if condition1:
    statement1
elif condition2:
    statement2
elif condition3:
    statement3
else:
    statement4
```

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。

猜一下以下程序打印的结果是什么？

In [ ]:
```
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：

In [113]:
```
x = 1
if x:
    print('True')
```

```
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：

In [115]:
```
birth = input('birth: ')
if birth < 2000:
    print('00前')
else:
    print('00后')
```

```
birth: 1982
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-115-e328fb0c508a> in <module>()
      1 birth = input('birth: ')
----> 2 if birth < 2000:
      3     print('00前')
      4 else:
      5     print('00后')

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：

In [119]:
```
s = input('birth: ')
birth = int(s)
if birth < 2000:
    print('00前')
else:
    print('00后')
```

```
birth: 1982
00前
```

但是，如果输入abc呢？

In [121]:
```
s = input('birth: ')
birth = int(s)
if birth < 2000:
    print('00前')
else:
    print('00后')
```

```
birth: abc

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-121-2064a032d405> in <module>()
      1 s = input('birth: ')
----> 2 birth = int(s)
      3 if birth < 2000:
      4     print('00前')
      5 else:

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

In [124]:
```
height = 1.75
weight = 80.5

bmi = weight / height**2

if bmi < 18.5:
    print('过轻')
elif bmi <= 25.0:
    print('正常')
elif bmi <= 28.0:
    print('过重')
elif bmi <= 32.0:
    print('肥胖')
else:
    print('严重肥胖')
```

```
过重
```

# 循环

要计算1 + 2 + 3，我们可以直接写表达式：

In [125]:
```
1 + 2 + 3
```

Out[125]: 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:

- for ... in
- while

## for ... in 循环

它可以依次把list或tuple中的每个元素迭代出来。

In [126]:
```
names = ['Michael', 'Bob', 'Tracy']
for name in names:
    print(name)
```

```
Michael
Bob
Tracy
```

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。

例：计算1-10的整数之和。

In [51]:
```
sum = 0
for x in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
    sum = sum + x
print(sum)
```

```
55
```

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。

range(5)生成0-4的整数序列：

In [59]:
```
list(range(5))
```

Out[59]: [0, 1, 2, 3, 4]

range(101)生成0-100的整数序列:

In [131]:
```
sum = 0
for x in range(101):
    sum = sum + x
print(sum)
```

```
5050
```

## while 循环

只要条件满足，就不断循环，条件不满足时退出循环。

例：计算100以内所有奇数之和。

In [132]:
```
sum = 0
n = 99
while n > 0:
    sum = sum + n
    n = n - 2
print(sum)
```

```
2500
```

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!：

In [135]:
```
L = ['Bart', 'Lisa', 'Adam']
```

## break

在循环中，break语句可以提前退出循环。

例如，循环打印1 ~ 100的数字：

In [ ]:
```
n = 1
while n <= 100:
    print(n)
    n = n + 1
print('END')
```

如果要提前结束循环，可以用break语句：

In [49]:
```
n = 1
while n <= 100:
    if n > 5: # 当n = 6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')
```

```
1
2
3
4
5
END
```

## 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。

例：输出数字1～6。

In [62]:
```
n = 0
while n < 6:
    n = n + 1
    print(n)
```

```
1
2
3
4
5
6
```

但是，如果我们想只打印奇数，可以用continue语句跳过某些循环。

In [63]:
```
n = 0
while n < 10:
    n = n + 1
    if n % 2 == 0:
        continue
    print(n)
```

```
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。