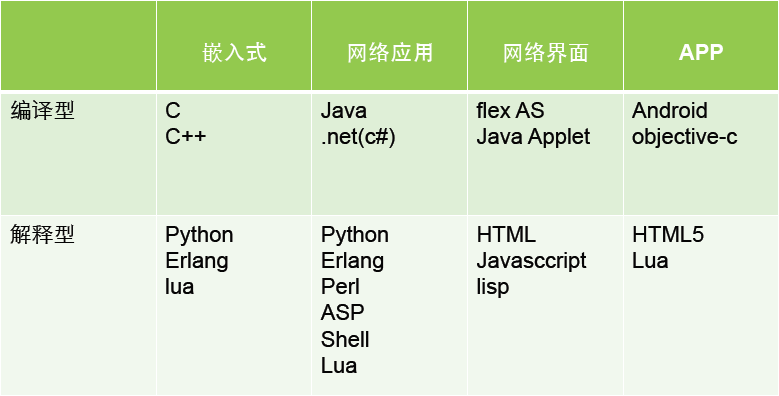
**Python笔记**

Python为弱类型语言，不强调编译时的类型，在运行时可以动态改变变量的类型。



Python函数在命名初期，可以给参数初始化默认数值，如下所示：

**def** say(message, times = 1):

**print**(message \* times)

只有在形参表末尾的那些参数可以有默认参数值，即你不能在声明函数形参的时候，先声明有默认值的形参而后声明没有默认值的形参。这是因为赋给形参的值是根据位置而赋值的。例如，def func(a, b=5) 是有效的，但是def func (a=5, b) 是无效的。

如果你的某个函数有许多参数，而你只想指定其中的一部分，那么你可以通过命名来为这些参数赋值—— 这被称作关键参数—— 我们使用名字（关键字）而不是位置（我们前面所一直使用的方法）来给函数指定实参。

这样做有两个优势—— 一、由于我们不必担心参数的顺序，使用函数变得更加简单了。二、假设其他参数都有默认值，我们可以只给我们想要的那些参数赋值。

**def** func(a, b=5, c=10):

**print**('a is', a, 'and b is', b, 'and c is', c)

func(3, 7)

func(25, c=24)

func(c=50, a=100)

输出为：

a is 3 and b is 7 and c is 10

a is 25 and b is 5 and c is 24

a is 100 and b is 5 and c is 50

文档字符串的惯例是一个多行字符串，它的首行以大写字母开始，句号结尾。第二行是空行，从第三行开始是详细的描述。强烈建议你在你的函数中使用文档字符串时遵循这个惯例。

你可以使用\_\_doc\_\_（注意双下划线）调用printMax 函数的文档字符串属性（属于函数的名称）。

一般说来，应该避免使用from..import 而使用import 语句，因为这样可以使你的程序更加易读，也可以避免名称的冲突。

name = input('please enter your name: ') ##可以提示输入

print('hello,', name)

/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，还有一种除法是地板除//,两个整数的除法仍然是整数。

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。

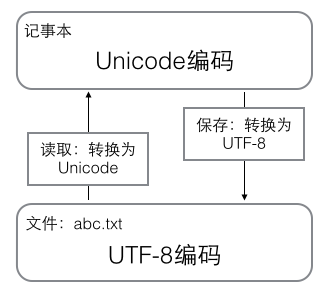
新的问题又出现了：如果统一成Unicode编码，乱码问题从此消失了。但是，如果你写的文本基本上全部是英文的话，用Unicode编码比ASCII编码需要多一倍的存储空间，在存储和传输上就十分不划算。

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：

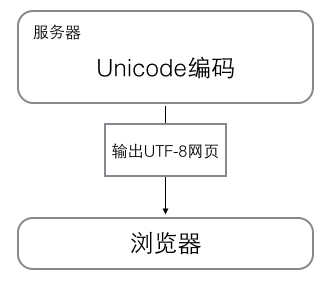
搞清楚了ASCII、Unicode和UTF-8的关系，我们就可以总结一下现在计算机系统通用的字符编码工作方式：

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：



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：

在Python中，采用的格式化方式和C语言是一致的，用%实现，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'

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。**list使用[]进行定义。**

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，**tuple使用()进行定义。**

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。

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。

后来看一个“可变的”tuple：

>>> t = ('a', 'b', ['A', 'B'])

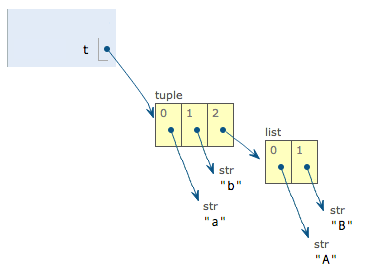
>>> t[2][0] = 'X'

>>> t[2][1] = 'Y'

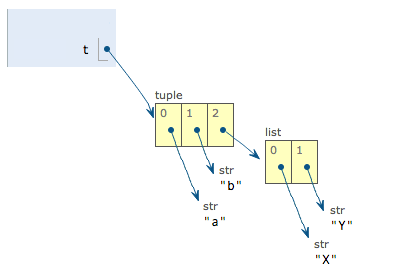
>>> t

('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！

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，**dict使用{}进行定义。**

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

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

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：

查找和插入的速度极快，不会随着key的增加而增加；

需要占用大量的内存，内存浪费多。

而list相反：

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。

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：

**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。

使用dict和set

424次阅读

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：

查找和插入的速度极快，不会随着key的增加而增加；

需要占用大量的内存，内存浪费多。

而list相反：

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。

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'

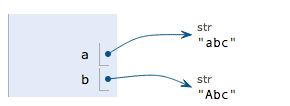
>>> a

'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'：**



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。

**函数定义**

**函数名其实就是指向一个函数对象的引用，完全可以把函数名赋给一个变量，相当于给这个函数起了一个“别名”：**

>>> a = abs # 变量a指向abs函数

>>> a(-1) # 所以也可以通过a调用abs函数

1

如果没有return语句，函数执行完毕后也会返回结果，只是结果为None。return None可以简写为return。

空函数

如果想定义一个什么事也不做的空函数，可以用pass语句：

def nop():

pass

pass语句什么都不做，那有什么用？实际上pass可以用来作为占位符，比如现在还没想好怎么写函数的代码，就可以先放一个pass，让代码能运行起来。

pass还可以用在其他语句里，比如：

if age >= 18:

pass

缺少了pass，代码运行就会有语法错误。

但其实这只是一种假象，Python函数返回的仍然是单一值：

**返回多个值**

**函数可以返回多个值吗？答案是肯定的。**

>>> r = move(100, 100, 60, math.pi / 6)

>>> print(r)

(151.96152422706632, 70.0)

原来返回值是一个tuple！但是，在语法上，返回一个tuple可以省略括号，而多个变量可以同时接收一个tuple，按位置赋给对应的值，所以，Python的函数返回多值其实就是返回一个tuple，但写起来更方便。

**函数默认参数的设置：**

Python的错误信息很明确：调用函数power()缺少了一个位置参数n。

这个时候，默认参数就排上用场了。由于我们经常计算x2，所以，完全可以把第二个参数n的默认值设定为2：

def power(x, n=2):

s = 1

while n > 0:

n = n - 1

s = s \* x

return s

从上面的例子可以看出，默认参数可以简化函数的调用。设置默认参数时，有几点要注意：

一是必选参数在前，默认参数在后，否则Python的解释器会报错（思考一下为什么默认参数不能放在必选参数前面）；

二是如何设置默认参数。

**当函数有多个参数时，把变化大的参数放前面，变化小的参数放后面。变化小的参数就可以作为默认参数。**

**使用默认参数有什么好处？最大的好处是能降低调用函数的难度。**

**可变参数的函数：**

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，如果没有使用可变参数，**上述函数需要传入list或者tuple，调用方式为calc([1,2,3]),使用可变参数后，调用方式可以变为calc（1,2,3）。**

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

**切片**

>>> L[0:3]

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

L[0:3]表示，从索引0开始取，直到索引3为止，但不包括索引3。即索引0，1，2，正好是3个元素。

如果第一个索引是0，还可以省略：

>>> L[:3]

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

可以通过切片轻松取出某一段数列。比如前10个数：

>>> L[:10]

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

后10个数：

>>> L[-10:]

[90, 91, 92, 93, 94, 95, 96, 97, 98, 99]

**生成器**

通过列表生成式，我们可以直接创建一个列表。但是，受到内存限制，列表容量肯定是有限的。而且，创建一个包含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，那后面绝大多数元素占用的空间都白白浪费了。

所以，如果列表元素可以按照某种算法推算出来，那我们是否可以在循环的过程中不断推算出后续的元素呢？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。

要创建一个generator，有很多种方法。第一种方法很简单，只要把一个列表生成式的[]改成()，就创建了一个generator：

>>> L = [x \* x for x in range(10)]

>>> L

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

>>> g = (x \* x for x in range(10))

>>> g

我们讲过，generator保存的是算法，每次调用next(g)，就计算出g的下一个元素的值，直到计算到最后一个元素，没有更多的元素时，抛出StopIteration的错误。

当然，上面这种不断调用next(g)实在是太变态了，正确的方法是使用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)

定义generator的另一种方法。如果一个函数定义中包含yield关键字，那么这个函数就不再是一个普通函数，而是一个generator：

**迭代器**

凡是可作用于for循环的对象都是Iterable类型；

凡是可作用于next()函数的对象都是Iterator类型，它们表示一个惰性计算的序列；

集合数据类型如list、dict、str等是Iterable但不是Iterator，不过可以通过iter()函数获得一个Iterator对象。

**函数名也是变量**

那么函数名是什么呢？函数名其实就是指向函数的变量！对于abs()这个函数，完全可以把函数名abs看成变量，它指向一个可以计算绝对值的函数！

既然变量可以指向函数，函数的参数能接收变量，那么一个函数就可以接收另一个函数作为参数，这种函数就称之为高阶函数。

一个最简单的高阶函数：

def add(x, y, f):

return f(x) + f(y)

if \_\_name\_\_=='\_\_main\_\_':

test()

当我们在命令行运行hello模块文件时，Python解释器把一个特殊变量\_\_name\_\_置为\_\_main\_\_，而如果在其他地方导入该hello模块时，if判断将失败，因此，这种if测试可以让一个模块通过命令行运行时执行一些额外的代码，最常见的就是运行测试。

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，所以，不能用\_\_name\_\_、\_\_score\_\_这样的变量名。

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”。

双下划线开头的实例变量是不是一定不能从外部访问呢？其实也不是。不能直接访问\_\_name是因为Python解释器对外把\_\_name变量改成了\_Student\_\_name，所以，仍然可以通过\_Student\_\_name来访问\_\_name变量：

任何模块代码的第一个字符串都被视为模块的文档注释；

软件方法论：

1. 笛卡尔方法论：拆分、排序、处理、归并
2. 软件工程过程方法论：
3. 软件声明周期
4. RUP统一软件过程管理
5. 敏捷开发
6. 项目管理（计算、组织、执行在、控制）

Python安装版本为2.7，然后需要安装setuptools工具，使用cmd窗口运行（以管理员权限运行cmd）python ez\_setup.py 如果这期间有报错编码错误，则需要修改C:\Python27\Lib下的mimetypes.py 文件中如下的代码：

default\_encoding = sys.getdefaultencoding()

在这行前面添加三行：

if sys.getdefaultencoding() != 'gbk':

    reload(sys)

    sys.setdefaultencoding('gbk')

default\_encoding = sys.getdefaultencoding()

然后即可以正确安装setuptools，然后将C:\Program Files\Python\Scripts加入到环境变量中，以下是使用setuptools安装django的例子

在cmd中运行easy\_install django

easy\_install类似于ubuntu中的apt-get install命令

python中单行注释使用#，多行注释是用三个单引号或者三个双引号

python中的常量需要使用常量修饰符。

python中字符串单引号与双引号的使用效果相同。

三引号的作用：也可以用来赋值字符串，如果三引号不出现在赋值的时候，三引号可以用来表示块注释。

a = r"adcd\nasdfa"

print a

字符串前加个r表示后面的字符串为原生类型字符串，即不会将\n转义为换行

eval

python中的数据类型：

* 数值
* 字符串
* 线性容器

-字符串也是一种线性容器

-List

-tuple

* Hash容器

-Dict

-set

* None
* 逻辑类型（True False）

for a in xrange(100):

print a

xrange效率高于range

python中没有++和—操作

python的while循环中有一种特殊的else用法：

a = 0

while a < 100:

print a

a = a + 1

if a > 50:

break

else:

print "over"

上述代码中的else只有while循环是因为break结束的才会执行，如果是正常结束的（将if中的内容去掉），则不会执行else中的内容。

**python中函数的定义：**

函数可以很方便的使用f1 = f对其进行别名的拷贝，调用f1和f的效果相同

函数可以返回多个值，也可以用多个变量去接收这个返回值，如果用一个变量接收多个返回值，则这个变量为tuple

lambda的特殊用法：

f1 = lamba x, y: x+y

print f1(2,3)

结果输出为5

**类中成员函数的定义：**

def \_\_init\_\_(self): //构造函数

def \_\_a(self) ://私有函数

def b(self): //共有函数

每个类内的函数都需要出现self 关键字

def c(self):

pass //pass关键字用来站位，对于还没想好需要写什么代码的地方可以使用pass代替。

**文件与包的导入：**

如果需要使用from test import test1.py从某个文件夹中导入相应的py文件，需要在该文件夹下放入一个\_\_init\_\_.py文件才可以完成调用。

**Python的容器：**

**List**

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

type(a)

**tuple**

a=(1,2,3,4,5)

type(a)

元组和列表的区别在于，元组只适合在声明之后不需要进行增删改的操作，(查询效率会比列表高很多)如果需要进行增删改的操作，需要将其先转换为列表之后再进行操作。

**字典**

a={“a”:1, “b”:2}

type(a)

**set**

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

b = set(a)

type(b)

map reduce filter操作：

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

map(lambda x:2\*x, a)

reduce(lambda x, y:x+y, a)

filter(lambda x: x%2, a)

迭代器

import os

b = os.walk(.)

b.next()

生成器（函数中包含yield关键字时即为生成器）

**正则表达式：**

import re

p = re.compile(r"abc")

m = p.match("abcdef")

print p.match("abcdef").group()

m = p.match("[abc]+") //贪婪模式，必须满足abc才会返回

m = p.match("[abc]+?") //非贪婪模式，只要有满足a, b, c任何一项即返回

m = p.match("[abc]{2,3}?") //非贪婪模式

边界符的使用：

m = p.match("^[abc]+?") //非贪婪模式,以a,b,c中任意一个字母作为开头

m = p.match("^[abc]+e$?") //非贪婪模式,以a,b,c中任意一个字母作为开头,且以e结尾

**m.group(), m.groups(), m.groupdict()**

**逻辑或者分组的概念：**

Python的几种常见的软件安装工具有：distribute,setuptools（使用easy\_install命令进行包的安装）与pip。推荐使用PIP。



easy\_install是由PEAK(Python Enterprise Application Kit)开发的setuptools包里带的一个命令，所以使用easy\_install实际上是在调用setuptools来完成安装模块的工作。

python -m pip install -U pip //windows中升级pip

在系统的环境变量中设置C:\Python34\Scripts，然后就可以在cmd窗口中使用pip命令。

对应的是esay\_install命令。

pip install pil //通过pip命令安装PIL插件。

**文本文件与二进制文件的区别**

大家都知道计算机的存储在物理上是二进制的，所以文本文件与二进制文件的区别并不是物理上的，而是逻辑上的。这两者只是在编码层次上有差异。简单来说，文本文件是基于字符编码的文件，常见的编码有ASCII编码，UNICODE编码等等。二进制文件是基于值编码的文件，你可以根据具体应用，指定某个值是什么意思（这样一个过程，可以看作是自定义编码。

从上面可以看出文本文件基本上是定长编码的(也有非定长的编码如UTF-8)。而二进制文件可看成是变长编码的，因为是值编码嘛，多少个比特代表一个值，完全由你决定。大家可能对BMP文件比较熟悉，就拿它举例子吧，其头部是较为固定长度的文件头信息，前2字节用来记录文件为BMP格式，接下来的8个字节用来记录文件长度，再接下来的4字节用来记录bmp文件头的长度。

二、文本文件与二进制文件的存取

文本工具打开一个文件的过程是怎样的呢？拿记事本来说，它首先读取文件物理上所对应的二进制比特流，然后按照你所选择的解码方式来解释这个流，然后将解释结果显示出来。一般来说，你选取的解码方式会是ASCII码形式（ASCII码的一个字符是８个比特），接下来，它8个比特8个比特地来解释这个文件流。例如对于这么一个文件流"01000000\_01000001\_01000010\_01000011"(下划线''\_''，为了增强可读性手动添加的)，第一个8比特''01000000''按ASCII码来解码的话，所对应的字符是字符''A''，同理其它3个8比特可分别解码为''BCD''，即这个文件流可解释成“ABCD”，然后记事本就将这个“ABCD”显示在屏幕上。

事实上，世界上任何东西要与其他东西通信会话，都存在一个既定的协议，既定的编码。人与人之间通过文字联络，汉字“妈”代表生你的那个人，这就是一种既定的编码。但注意到这样一种情况，汉字“妈”在日本文字里有可能是你生下的那个人，所以当一个中国人Ａ与日本Ｂ之间用“妈”这个字进行交流，出现误解就很正常的。用记事本打开二进制文件与上面的情况类似。记事本无论打开什么文件都按既定的字符编码工作（如ASCII码），所以当他打开二进制文件时，出现乱码也是很必然的一件事情了，解码和译码不对应嘛。例如文件流''00000000\_00000000\_00000000\_00000001''可能在二进制文件中对应的是一个四字节的整数int 1，在记事本里解释就变成了"NULL\_NULL\_NULL\_SOH"这四个控制符。

WingIDE快捷键:

F1、F2可以隐藏下部和右侧的窗口

Tab键自动排版

Python3大法好，天灭str，退Python2保平安。   
人在做，天在看，混淆字符字节留隐患。   
参数乱传天地灭，函数标注保平安。   
诚心诚念from import好，Python3大法平安保。   
众生都为Enum来，现世险恶忘前缘。   
Python3弟子说真相，教你asyncio莫拒绝。   
早日摆脱u"xxxxx"，早日获得新生。   
上网搜“九评Python2”有真相。