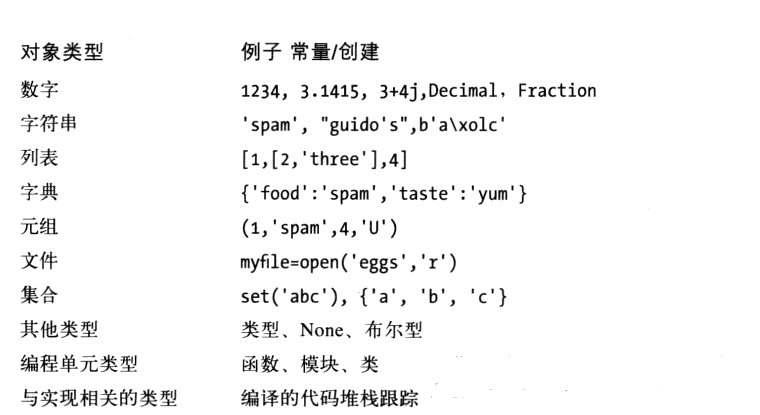
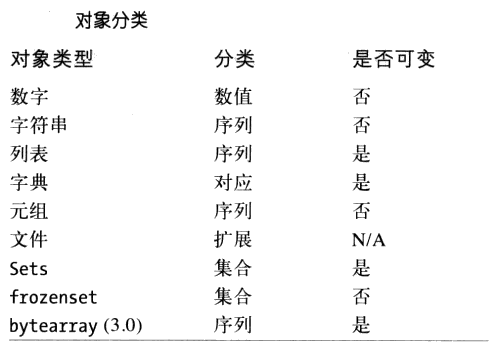
## 1.Python核心数据类型



Python是动态类型的（它自动的跟踪你的类型而不是要求声明代码），但是它也是强类型语言（你只能对一个对象进行适合该类型的有效的操作）。

## 变与不变



python中不存在所谓的传值调用，**一切传递的都是对象的引用**，也可以认为是传址。python中，对象分为**可变(mutable)**和不可变(immutable)两种类型，**元组（tuple)、数值型（number)、字符串(string)均为不可变对象,而字典型(dictionary)和列表型(list)的对象是可变对象。**

**可变即内存空间可变，不可变即内存空间不可变。** 可变的对象的变量在传递时，传递的是引用地址（可以理解为指针），所以拷贝的时候不是真正拷贝内存空间，仅仅是多了个变量指向相同的内存空间而已。对新的变量进行增删都会影响到原始变量，即浅拷贝。不可变对象，拷贝或通过新变量传递时，一旦发生修改，会新产生一个跟原来对象属于不同内存空间的新对象。

### 2.1 不可变性

>>> a = 1 #将名字a与内存中值为1的内存绑定在一起

>>> a = 2 #将名字a与内存中值为2的内存绑定在一起，而不是修改原来a绑定的内存的值，这时，内存中值为1的内存地址引用计数为-1，当引用计数为0时，内存被收回

>>> b = a #变量b执行与a绑定的内存

>>> b = 3 #将名字b与内存中值为3的内存绑定在一起，这时a还是指向内存为2的地址

>>> a,b

(2, 3)

字符串所包含的字符存在从左到右的位置顺序，并且它们不可以在原处修改。

>>> S = 'spam' #将名字S与内存中值为’spam’的内存绑定在一起

>>> S[0] = 'z' #不可变对象不支持通过对字符串某一位置进行赋值来改变对象

Traceback (most recent call last):

File "<pyshell#6>", line 1, in <module>

S[0] = 'z'

TypeError: 'str' object does not support item assignment

>>> S = 'z'+S[1:] #将名字S与一个表达式生成的新的对象内存绑定在一起

>>> S

'zpam'

不可变类型不可以在原处改变

>>> T = (1,2,3)

>>> T[2] = 4

Traceback (most recent call last):

File "<pyshell#22>", line 1, in <module>

T[2] = 4

TypeError: 'tuple' object does not support item assignment

>>> T = T[:2]+(4,)

>>> T

(1, 2, 4)

**python对不可变对象，通过构造函数和[:]创建的副本，是对同一对象的引用。**

>>> t1 = (1,2,3)

>>> t2 = tuple(t1)

>>> t1 is t2

True

>>> t3 = t1[:]

>>> t3 is t1

True

### 2.2 可变性

对于列表和字典支持在原处修改（它们是可变的）。

Python列表包含了零个或多个其他对象的引用。当把一个对象赋给一个数据结构元素或变量名时，python总会存储对象的引用，而不是对象的一个拷贝。由于列表是可变的，它们支持原处改变列表对象的操作。

>>> L = ['spam', 'Spam', 'SPAM!']

>>> L[1] = 'eggs'

>>> L

['spam', 'eggs', 'SPAM!']

>>> L[0:2] = ['eat', 'more']

>>> L

['eat', 'more', 'SPAM!']

>>>

索引和分片的赋值都是原地修改，它们对列表进行直接修改，而不是生成一个新的列表作为结果。

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

>>> M = L

>>> M

[1, 2, 3]

>>> L.append(4)

>>> M

[1, 2, 3, 4]

>>> L.extend([6,5])

>>> M

[1, 2, 3, 4, 6, 5]

>>> L.sort()

>>> M

[1, 2, 3, 4, 5, 6]

>>> L += [7, 8]

>>> M

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

>>> L = L + [9, 10]

>>> M

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

>>> L

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>>>

列表的append、extend、sort方法都是对列表原地修改。“+=”对列表是原处修改的意思，“+”合并总是生成新的对象。

>>> D0 = {'eggs':3, 'ham':1}

>>> D1 = D0

>>> D1

{'eggs': 3, 'ham': 1}

>>> D0['ham'] = [1,2,3]

>>> D1

{'eggs': 3, 'ham': [1, 2, 3]}

>>> D0['apples'] = 5

>>> D1

{'eggs': 3, 'ham': [1,2,3], 'apples': 5}

>>>

与列表相同，向字典中已存在的索引赋值会改变与索引相关联的值。每当对新字典键进行赋值（之前没有赋值的键），就会在字典内生成新的元素。

>>> D2 = {'toast':4, 'muffin':5}

>>> D0.update(D2)

>>> D1

{'toast': 4, 'muffin': 5, 'eggs': 3, 'ham': [1, 2, 3], 'apples': 5}

>>> D0.pop('toast')

4

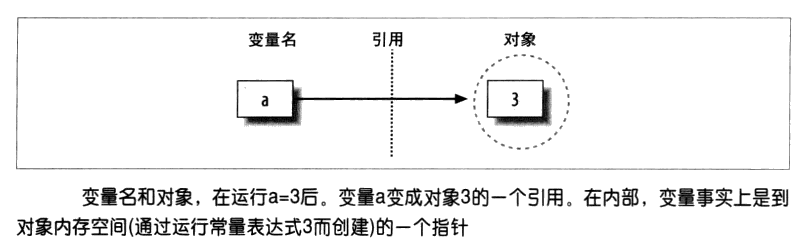
>>> D1

{'muffin': 5, 'eggs': 3, 'ham': [1, 2, 3], 'apples': 5}

>>>

## 动态类型

### 3.1 变量、对象和引用



变量在赋值的时候才创建，它可以引用任何类型的对象，并且必须在引用之前赋值。变量和对象保存在内存中不同部分，并且通过连接相关联。**变量总是连接到对象，并且绝不会连接到其它变量上**，但是更大的对象可能连接到其它对象（例如，一个列表对象）。

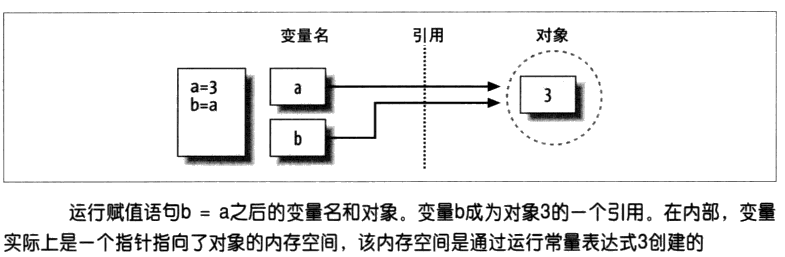
在python中从变量到对象的连接称为引用。引用是一种关系，以内存中的指针的形式实现。

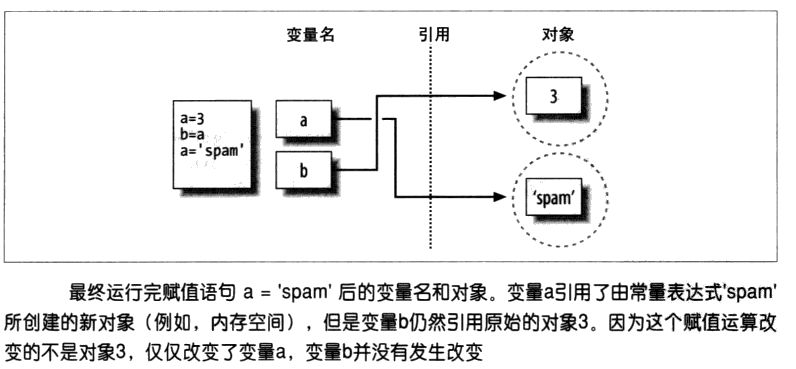
* 变量是一个系统表的元素，拥有指向对象的连接的空间。
* 对象是分配的一块内存，有足够的空间去表示它们所代表的值。每一个对象都有两个标准的头部信息：一个类型标志符去标识这个对象的类型，以及一个引用的计数器。
* 引用是自动形成的从变量到对象的指针。

变量名没有类型，类型属于对象而不是变量名。**Python中的类型是与对象相关联的，而不是和变量关联。（与C语言等不同）**

**对象的垃圾收集：每个对象中保持了一个计数器，计数器记录了当前指向该对象的引用的数目。一旦（并精确在同一时间）这个计数器被设置为零，这个对象的内存空间就会自动回收。**

### 3.2 共享引用





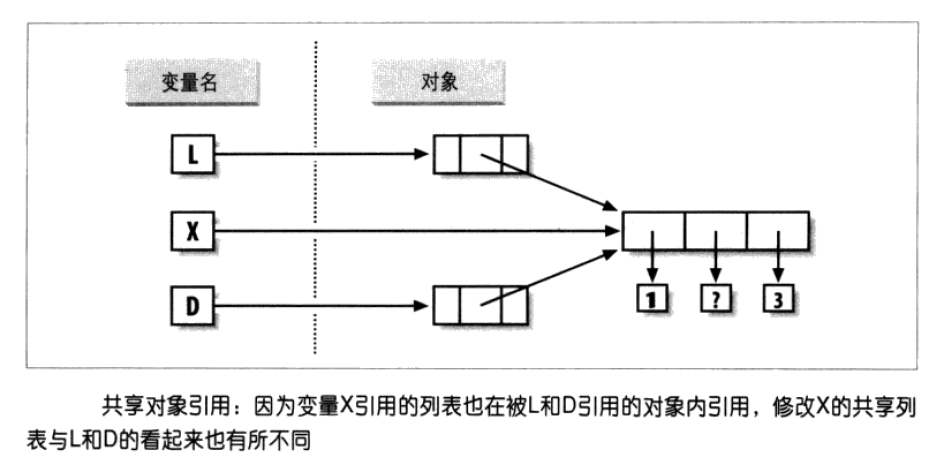
在python中，变量总是一个指向对象的指针，而不是可改变的内存区域的标签：给一个变量赋一个新的值，并不是替换原始的对象，而是让这个变量去引用完全不同的一个对象。

## 引用VS拷贝

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

>>> L = ['a', X, 'b']

>>> D = {'x':X, 'y':2}



>>> X[1] = 'surprise'

>>> L

['a', [1, 'surprise', 3], 'b']

>>> D

{'y': 2, 'x': [1, 'surprise', 3]}

>>>

* 没有限制的分片表达式（L[:]）能够复制序列
* 字典copy方法（X.copy()）能够复制字典
* 有些内置函数（例如，list）能够生成拷贝（list（L））
* copy标准库模块能够生成完整拷贝

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

>>> L= ['a', X[:], 'b']

>>> D = {'x':X[:], 'y':2}

>>> l1 = [3, [55, 44], (7, 8, 9)]

>>> l2 = list(l1)

>>> l2

[3, [55, 44], (7, 8, 9)]

>>> l2 == l1

True

>>> l2 is l1

False

>>> l3 = l1[:]

>>> l3

[3, [55, 44], (7, 8, 9)]

>>> l3 == l1

True

>>> l3 is l1

False

**构造方法或[:]做的是浅复制（即复制了最外层的容器，副本中的元素是源容器中元素的引用）。**

无条件的分片以及字典copy方法只能做顶层复制。如果需要一个深层嵌套的数据结构的完整的、完全独立的拷贝，那么就要使用标准copy模块，X = copy.deepcopy（Y）对任意嵌套对象Y做完整的复制。

## 比较、相等性

* “==”操作符测试值的相等性
* “is”表达式测试对象的一致性

‘is’相对来说效率更高，它用来比较对象的id值，而“==”则会调用\_\_eq\_\_重载运算符。

## 函数参数和共享引用

* 不可变参数“通过值”进行传递
* 可变对象是通过“指针”进行传递

>>> def f(a):

a = 99

>>> b = 88

>>> f(b)

>>> print b

88

>>>

在python中，默认通过引用（也就是指针）进行函数的参数传递。即python中唯一支持的参数传递模式是共享传参（call by sharing）。

>>> def changer(a, b):

a = 2

b[0] = 'spam'

>>> X = 1

>>> L = [1, 2]

>>> changer(X, L)

>>> X, L

(1, ['spam', 2])

### 不要使用可变类型作为参数的默认值

class **HauntedBus**:

*"""备受幽灵乘客折磨的校车 """*

def **\_\_init\_\_**(*self*, passengers=[]):

*self*.passengers = passengers

def **pick**(*self*, name):

*self*.passengers.append(name)

def **drop**(*self*, name):

*self*.passengers.remove(name)

def **test\_hauntedbus**():

bus1 = HauntedBus([*'Alice'*, *'Bill'*])

print bus1.passengers # ['Alice', 'Bill']

bus1.pick(*'Charlie'*)

bus1.drop(*'Alice'*)

print bus1.passengers # ['Bill', 'Charlie']

bus2 = HauntedBus()

bus2.pick(*'Carrie'*)

print bus2.passengers # ['Carrie']

bus3 = HauntedBus()

print bus3.passengers # ['Carrie']

bus3.pick(*'Dave'*)

print bus2.passengers # ['Carrie', 'Dave']

print bus2.passengers is bus3.passengers # True

passengers默认值引用了同一个可变对象。

### 防御可变参数

class **TwilightBus**:

*"""让乘客销声匿迹的校车 """*

def **\_\_init\_\_**(*self*, passengers=None):

if passengers is None:

*self*.passengers = []

else:

*self*.passengers = passengers

def **pick**(*self*, name):

*self*.passengers.append(name)

def **drop**(*self*, name):

*self*.passengers.remove(name)

def **test\_twilightbus**():

basketball\_team = [*'Sue'*, *'Tina'*, *'Maya'*, *'Diana'*, *'Pat'*]

bus = TwilightBus(basketball\_team)

bus.drop(*'Tina'*)

bus.drop(*'Pat'*)

print bus.passengers # ['Sue', 'Maya', 'Diana']

**TwilightBus违反了接口设计的最佳实践，即“最少惊讶原则”。学生从校车中下车后，**

**她的名字就从篮球队的名单中消失了，这确实让人惊讶。**

## 集合

集合（set）是一些唯一的、不可变对象的一个无序集合。集合本身是**可变的**。

>>> id(s)

50295080L

>>> s.update('efg')

>>> id(s)

50295080L

**集合只能包含不可变的对象类型**，因此，列表和字典不能嵌入到集合中，元组是可以嵌入的。

set()：创建一个新的set对象。

>>> x = set('abcde')

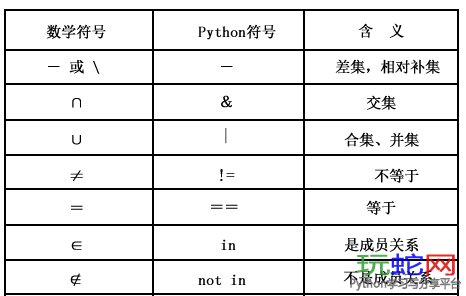
>>> x

set(['a', 'c', 'b', 'e', 'd'])

>>> y = set('bdefg')

>>> y

set(['b', 'e', 'd', 'g', 'f'])



**判断是否在集合中**：

>>> 'a' in x

True

**差集**：

>>> x - y

set(['a', 'c'])

>>> x + y

**并集**：

>>> x | y

set(['a', 'c', 'b', 'e', 'd', 'g', 'f'])

**交集**：

>>> x & y

set(['b', 'e', 'd'])

**XOR异或**：

>>> x ^ y

set(['a', 'c', 'g', 'f'])

**比较：**

>>> x > y, x < y, x == y, x!= y

(False, False, False, True)

### set方法

**add():**

set中添加一个元素。

>>> x.add('f')

>>> x

set(['a', 'c', 'b', 'e', 'd', 'f'])

**clear():**

清空set

>>> x.clear()

>>> x

set([])

**copy():**

返回一个set浅拷贝。

>>> x = set('abc')

>>> x

set(['a', 'c', 'b'])

>>> z = x.copy()

>>> z

set(['a', 'c', 'b'])

**difference(...)**：

#!/usr/bin/env python

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

# 集合是一个唯一的不可变对象的集合。

# 基本操作

# 创建集合

# 空集合

var = set()

print var , type(var) # set([]) <type 'set'>

# 具有数据的集合

var = {*'zhangwang'*,*'zhangbo'*,*'zhanglang'*}

print var, type(var) # set(['zhangbo', 'zhangwang', 'zhanglang']) <type 'set'>

# 成员检测

result = *'zhangbo'* in var

print result # True

result = *'zhangsan'* not in var

print result # True

# 集合推导式

var = {*'蚂蚱'*,*'螳螂'*,*'蝈蝈'*,*'蛐蛐'*}

# 基本的集合推导式

result = {*'\*'*+i+*'\*'* for i in var}

print result

# 带有判断条件的集合推导式

result = {i for i in var if i != *'蛐蛐'*}

print result

# 多循环集合推导式

colors = {*'red'*, *'blue'*, *'pink'*}

sizes = {36, 37, 38, 39}

result = {c + str(s) for c in colors for s in sizes}

print result # set(['blue39', 'blue38', 'blue37', 'pink37', 'blue36', 'pink36', 'red39', 'red38', 'pink38', 'pink39', 'red37', 'red36'])

# 集合函数

*'''*

*# add() 向集合中添加元素*

*girls = {'mf','sl','yj'}*

*girls.add('xf')*

*print girls*

*# pop() 随机删除集合中的一个元素*

*boys = {'bd','zw','jl','zy'}*

*result = boys.pop()*

*print boys*

*print result*

*# remove() 删除集合中指定的元素 删除不存在的元素会报错*

*boys = {'bd','zw','jl','zy'}*

*boys.remove('zy')*

*print boys*

*# discard() 删除集合中指定的元素 删除不存在的元素啥都不做*

*boys = {'bd','zw','jl','zy'}*

*boys.discard('zy1')*

*print boys*

*# clear() 清空集合*

*boys = {'bd','zw','jl','zy'}*

*boys.clear()*

*print boys*

*# copy() 复制集合*

*boys = {'bd','zw','jl','zy'}*

*newboys = boys.copy()*

*print newboys*

*'''*

# difference() 返回一个新的集合，包含集合的差集

dreamers = {*'ljl'*,*'wc'*,*'xy'*,*'zb'*,*'lsy'*}

girls = {*'mmf'*,*'lsy'*,*'syj'*}

result = dreamers.difference(girls) # result = a + b

print result # set(['xy', 'wc', 'ljl', 'zb'])

#difference\_update() 计算多个集合的差集(差集更新操作)

dreamers = {*'ljl'*,*'wc'*,*'xy'*,*'zb'*,*'lsy'*}

girls = {*'mmf'*,*'lsy'*,*'syj'*}

dreamers.difference\_update(girls) # a = a + b a += b

print dreamers # set(['xy', 'wc', 'ljl', 'zb'])

# union() 并集操作

dreamers = {*'ljl'*,*'wc'*,*'xy'*,*'zb'*,*'lsy'*}

girls = {*'mmf'*,*'lsy'*,*'syj'*}

result = dreamers.union(girls)

print result # set(['wc', 'ljl', 'syj', 'zb', 'xy', 'mmf', 'lsy'])

# update() 并集更新操作，用自身或其他集合更新一个集合。

dreamers = {*'ljl'*,*'wc'*,*'xy'*,*'zb'*,*'lsy'*}

girls = {*'mmf'*,*'lsy'*,*'syj'*}

dreamers.update(girls) # set(['wc', 'ljl', 'syj', 'zb', 'xy', 'mmf', 'lsy'])

print dreamers

x = {1, 2, 3}

x.update(*'abc'*) # 等价于x.update(set('abc'))

print x # set(['a', 1, 2, 3, 'c', 'b'])

# intersection() 计算2个集合的交集

dreamers = {*'ljl'*,*'wc'*,*'xy'*,*'zb'*,*'lsy'*}

girls = {*'mmf'*,*'lsy'*,*'syj'*}

result = dreamers.intersection(girls)

print result # set(['lsy'])

# intersection\_update 交集更新操作

dreamers = {*'ljl'*,*'wc'*,*'xy'*,*'zb'*,*'lsy'*}

girls = {*'mmf'*,*'lsy'*,*'syj'*}

dreamers.intersection\_update(girls)

print dreamers # set(['lsy'])

# 超集和子集

boys = {*'zzy'*,*'yqw'*,*'dw'*,*'wzc'*,*'lyb'*,*'wym'*,*'chy'*}

zoudu = {*'wzc'*,*'lyb'*,*'wym'*}

girls = {*'lsy'*,*'mmf'*,*'syj'*}

# issuperset() 检测当前集合是否是另一个集合的超集

result = boys.issuperset(zoudu)

print result # True

# issubset() 检测当前集合是否是另一个集合的子集

result = zoudu.issubset(boys)

print result # True

# isdisjoint() 检测2个集合是否不存在交集 存在交集 False

result = boys.isdisjoint(girls)

print result # True

# symmetric\_difference() 对称差集

dreamers = {*'ljl'*,*'wc'*,*'xy'*,*'zb'*,*'lsy'*}

girls = {*'mmf'*,*'lsy'*,*'syj'*}

result = dreamers.symmetric\_difference(girls)

print result

#symmetric\_difference\_update() 对称更新差集

dreamers = {*'ljl'*,*'wc'*,*'xy'*,*'zb'*,*'lsy'*}

girls = {*'mmf'*,*'lsy'*,*'syj'*}

dreamers.symmetric\_difference\_update(girls)

print(dreamers)

# 冰冻集合

# 冰冻集合是一种特殊的集合类型,也是集合(集合是列表的话,冰冻集合就是元组)

# 创建冰冻集合

# 一般不会创建空的冰冻集合

var = frozenset()

print(var,type(var))

# 带有数据的冰冻集合

var = frozenset((*'qs'*,*'szx'*,*'bjh'*,*'acs'*))

print(var,type(var))

# 成员检测

result = *'szx'* in var

print(result)

# 遍历冰冻集合

for i in var:

print(i)

# 集合推导式(无法得到冰冻集合,可以得到集合,列表,元组,字典类型)

result = {i for i in var}

print(result,type(result))

# 函数

# 冰冻集合可以使用集合的函数(不修改集合本身的函数都可以使用)

var = frozenset((*'qs'*,*'szx'*,*'bjh'*,*'acs'*))

# copy()

result = var.copy()

print(result)

# 集合操作 交集,并集,差集,对称差集等 不修改冰冻集合本身就能使用:冰冻集合的操作结果都是冰冻集合

var1 = frozenset((*'qs'*,*'szx'*,*'bjh'*,*'acs'*))

var2 = {*'szx'*,*'bjh'*,*'lc'*,*'wb'*}

# 冰冻集合操作

result = var1.union(var2)

print(result)

# 普通集合操作(冰冻集合是参考集合)

result = var2.union(var1)

print(result)