高级程序设计
---Python与深度学习
3. 列表与字典
3. List and dictionary

李冰
副研究员
交叉科学研究院

课程内容

- 列表
  - 列表的基本操作
  - 列表常用操作
- 字典
  - 字典的基本操作
  - 字典常用操作
- 动态类型

列表

- 任意对象的有序集合
  - 从功能上看，列表就是收集其它对象的地方。同事列表所包含的每一项都保持了从左到右的位置顺序（也就是说它们是序列）。
- 通过偏移读取
  - 就像字符串一样，你可以通过列表对象的偏移对其进行索引，从而读取对象的某一部分内容。
- 可变长度、异构以及任意嵌套
  - 列表是可变长度的
  - 可以包含任意类型的对象（字符串只能包含单个字符）。
  - 支持在原处的修改，也可以响应所有针对字符串序列的操作，
  - 索引、分片以及合并。
  - 支持在原处的删除和索引赋值操作。

列表 --- 基本操作

- 列表是序列，它支持很多与字符串相同的操作。
  - 例如，列表对 + 和 * 操作的响应与字符串很相似，产生的结果是一个新的列表

```
L = []              # Empty list
L
```

```
[]
```

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

```
[1, 2, 3]
```

列表 --- 基本操作

- 列表是序列，它支持很多与字符串相同的操作。
  - 例如，列表对 + 和 * 操作的响应与字符串很相似，产生的结果是一个新的列表

```
[1, 2, 3] + [4, 5, 6]    # Concatenation
```

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

```
[4, 5, 6] * 4    # Repetition
```

```
[4, 5, 6, 4, 5, 6, 4, 5, 6, 4, 5, 6]
```

```
3 in [1, 2, 3]    # Membership
```

```
True
```

  - 索引

```
L = ['spam', 'Spam', 'SPAM!']    # Offsets start at zero
L[2], L[1:], L[-2]
```

列表迭代和解析

- for 循环从左到右遍历序列中的每一项，对每一项执行一条或多条语句。

```
for x in [1, 2, 3]:     # Iteration
    print(x, end=' ')
```

```
1 2 3
```

- 闭包语法

```
# 新的列表=[对元素的处理 for 元素 in 列表]
res = [c * 4 for c in 'SPAM']     # List comprehensions
res
```

```
['SSSS', 'PPPP', 'AAAA', 'MMMM']
```

分片和矩阵

- 由于列表是可变的，它们支持原处改变列表对象的操作。可以将一个特定项或整个片段来改变列表的内容。

```
L = ['spam', 'Spam', 'SPAM!']
L[1] = 'eggs'
L
```

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

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

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

- 可以用嵌套列表来表示矩阵，下面一个基于列表的3x3的二维数组。

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(matrix[1])
print(matrix[2][0])
```

```
[4, 5, 6]
7
```

列表操作

- Python 列表对象支持特定类型方法调用，其中很多方法可以在原处修改主体列表。

```
L = ['eat', 'more', 'SPAM!']
L.append('please')      # Append method call: add item at end
L
```

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

```
L.sort()                # Sort list items ('S' < 'e')
L
```

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

```
L.sort(reverse=True)
print(L)
```

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

注意：append() 和 sort() 都是原处修改列表，两个函数的返回值都是 None：

```
dummy = L.sort(reverse=True)
print(dummy)
```

```
None
```

列表操作-续

```
L = [1, 2]
L.extend([3, 4, 5])     # extend 在列表末端插入多个元素
L
```

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

```
print(L.pop())          # pop 弹出列表最后一个元素，可以与 append 方法联用，用来实现栈结构
print(L)
```

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

```
L.reverse()             # reverse 原地反转列表
L
```

```
[4, 3, 2, 1]
```

列表操作-续

```
L.insert(1, 'toast')  # Insert at position
L
```

```
['spam', 'toast', 'eggs', 'ham']
```

```
L.remove('eggs')      # Delete by value
L
```

```
['spam', 'toast', 'ham']
```

```
L.pop(1)              # Delete by position
```

```
'toast'
```

列表操作-续

```
L.insert(1, 'toast')  # Insert at position
L
```

```
['spam', 'toast', 'eggs', 'ham']
```

```
L.remove('eggs')      # Delete by value
L
```

```
['spam', 'toast', 'ham']
```

```
L.pop(1)              # Delete by position
```

```
'toast'
```

```
L
```

```
['spam', 'ham']
```

列表操作-续

```
L.insert(1, 'toast')  # Insert at position
L
```

```
['spam', 'toast', 'eggs', 'ham']
```

```
L.remove('eggs')      # Delete by value
L
```

```
['spam', 'toast', 'ham']
```

```
L.pop(1)              # Delete by position
```

```
'toast'
```

```
L
```

```
['spam', 'ham']
```

```
L.count('spam')       # Number of occurrences
```

```
1
```

```
L.clear()             # 清空列表
```

列表操作-小结

方法	例子及含义
L.append()	L.append(‘please‘) 末尾增加一项
L.sort(), L.sort(reverse=True)	L.sort() 默认递增顺序
L.index()	L.index(‘eggs’) ,返回某个已知元素的索引
L.extend()	L.extend([3, 4, 5])末尾增加多项
L.remove()	L.remove(‘eggs’) 按元素删除
L.insert()	L.insert(1, ‘toast‘) 按位置插入
L.pop()	L.pop()， L.pop(1) 弹出第i个/最后一个元素
L.count()	L.count(‘spam‘) 重复出现的次数
L.clear()	清空列表

课程内容

- 列表
  - 列表的基本操作
  - 列表常用操作
- 字典
  - 字典的基本操作
  - 字典常用操作
- 动态类型

字典(Dictionary)

- 字典是无序的集合
  - 与列表的区别：字典中的元素是通过键来存取的，而不是通过偏移存取。
- 通过键而不是偏移量来存取，实现快速搜索
  - 稀疏数据结构常用这种集合方式
  - 字典有时也叫 hash 表
    - 通过键将一系列值联系起来，采用键作为索引从字典中获取内容。
- 可变长、异构、任意嵌套
  - 字典可以在原处增长或是缩短
  - 字典元素可以是不同类型
  - 字典嵌套字典

字典(Dictionary)

```
D = {'spam': 2, 'ham': 1, 'eggs': 3}      # Make a dictionary
D['spam']                                 # Fetch a value by key
```

```
2
```

```
D
```

```
{'spam': 2, 'ham': 1, 'eggs': 3}
```

在这里，字典被赋值给一个变量D，键 'spam' 的值为整数2。
字典用键对其进行索引操作，这也意味着用键来读取，而不是用位置来读取。

字典(Dictionary)

```
D = {'spam': 2, 'ham': 1, 'eggs': 3}      # Make a dictionary
D['spam']                                 # Fetch a value by key
```

```
2
```

```
D
```

```
{'spam': 2, 'ham': 1, 'eggs': 3}
```

在这里，字典被赋值给一个变量D，键 'spam' 的值为整数2。
字典用键对其进行索引操作，这也意味着用键来读取，而不是用位置来读取。

```
D2 = {} # Assign by keys dynamically
D2['name'] = 'Bob'
D2['age'] = 40
D2
```

```
{'name': 'Bob', 'age': 40}
```

如果需要动态地创建字典，可以先构造一个空字典，然后逐一赋值。

字典(Dictionary)基本操作

- **len()** 返回存储在字典中的键值对数目
- **in** 成员关系表达式提供了键存在与否的测试方法
- **keys()** 方法能够返回字典中所有的键，将它们收集在一个列表中

```
len(D)                          # Number of entries in dictionary
```

```
3
```

```
'ham' in D                      # Key membership test alternative
# D.has_key('ham')              # Deprecated
```

```
True
```

```
list(D.keys())                  # Create a new list of D's keys
```

```
['spam', 'ham', 'eggs']
```

修改字典

- 简单地给一个键赋值就可以改变或者生成元素。

```
D = {'spam': 2, 'ham': 1, 'eggs': 3}
D['ham'] = ['grill', 'bake', 'fry']      # Change entry (value=list)
D
```

```
{'spam': 2, 'ham': ['grill', 'bake', 'fry'], 'eggs': 3}
```

```
del D['eggs']                            # Delete entry
D
```

```
{'spam': 2, 'ham': ['grill', 'bake', 'fry']}
```

```
D['brunch'] = 'Bacon'                    # Add new entry
D
```

```
{'spam': 2, 'ham': ['grill', 'bake', 'fry'], 'brunch': 'Bacon'}
```

每当新字典键进行赋值（之前没有被赋值的键），就会在字典内生成一个新的元素。

字典常用操作

```
dir(dict)
```

```
['__class__',
 '__class_getitem__',
 '__contains__',
 '__delattr__',
 '__delitem__',
 '__dir__',
 '__doc__',
 '__eq__',
 'clear',
 'copy',
 'fromkeys',
 'get',
 'items',
 'keys',
 'pop',
 'popitem',
 'setdefault',
 'update',
 'values']
```

values() 和 **items()** 方法分别返回字典的值列表和 (key, value) 键值对

```
D = {'spam': 2, 'ham': 1, 'eggs': 3}
list(D.values())
```

```
[2, 1, 3]
```

```
list(D.items())
```

```
[('spam', 2), ('ham', 1), ('eggs', 3)]
```

字典常用操作

- **get()** 方法可以用来读取键值。

```
D.get('spam')                    # A key that is there
```

```
2
```

```
print(D.get('toast'))            # A key that is missing
```

```
None
```

```
D.get('toast', 8)
```

```
8
```

如果键不在字典中返回默认值 **None** 或者设置的默认值。

字典常用操作

- **update()** 方法有点类似于合并，它把一个字典的键和值合并到另一个字典中，覆盖相同键的值。

```
D = {'spam': 2, 'ham': 1, 'eggs': 3}
D2 = {'toast':4, 'muffin':5}     # Lots of delicious scrambled order here
D.update(D2)
D
```

```
{'spam': 2, 'ham': 1, 'eggs': 3, 'toast': 4, 'muffin': 5}
```

不支持“+”，“*”操作

字典常用操作

- 字典 **pop()** 方法能够从字典中删除一个键并返回它的值，类似于列表的 pop() 方法。

```
D
```

```
{'spam': 2, 'ham': 1, 'eggs': 3, 'toast': 4, 'muffin': 5}
```

```
D.pop('muffin')          # pop a dictionary by key
```

```
5
```

```
D
```

```
{'spam': 2, 'ham': 1, 'eggs': 3, 'toast': 4}
```

字典常用操作

- 使用 for 循环对字典进行遍历。

下面的例子能够生成一个表格，把程序语言名称（键）映射到它们的作者（值）。可以通过语言名称索引来读取作者的名字：

```
table = {'Python' : 'Guido van Rossum',
         'Perl':     'Larry Wall',
         'Tcl':      'John Ousterhout'}
for language in table:
    print(language, '\t', table[language])
```

```
Python    Guido van Rossum
Perl      Larry Wall
Tcl       John Ousterhout
```

字典常用操作-小结

方法	例子及含义
D.keys()	返回键列表
D.value()	返回值列表
D.items()	返回键值对列表
D.get()	读取键对应的值，如果没有返回None或默认值
D.update()	合并字典
D.pop()	D.pop('key') 删除某个键值对，返回键对应的值
D.clear()	清空字典

字典模拟列表

```
D = {}
D[99] = 'spam'
print(D[99])
print(D)
```

```
spam
{99: 'spam'}
```

```
L = []
L[99] = 'spam'
```

```
IndexError: list assignment index out of range
```

字典用于稀疏数据结构

- 例如，多维数组中只有少数位置上有非零值：

```
Matrix = {}
Matrix[(2, 3, 4)] = 88
Matrix[(7, 8, 9)] = 99
Matrix
```

```
{(2, 3, 4): 88, (7, 8, 9): 99}
```

```
X = 2; Y = 3; Z = 4                          # ; separates statements
Matrix[(X, Y, Z)]
```

```
88
```

动态初始化字典

- **zip()** 函数

```
D = dict(zip(['a', 'b', 'c'], [1, 2, 3]))      # Make a dict from zip result
D
```

```
{'a': 1, 'b': 2, 'c': 3}
```

```
D = {k: v for (k, v) in zip(['a', 'b', 'c'], [1, 2, 3])}
D
```

```
{'a': 1, 'b': 2, 'c': 3}
```

字典的闭包语法

```
D = {x: x ** 2 for x in [1, 2, 3, 4]}          # Or: range(1, 5)
D
```

```
{1: 1, 2: 4, 3: 9, 4: 16}
```

```
D = {c: c * 4 for c in 'SPAM'} # Loop over any iterable
D
```

```
{'S': 'SSSS', 'P': 'PPPP', 'A': 'AAAA', 'M': 'MMMM'}
```

字典用法注意事项

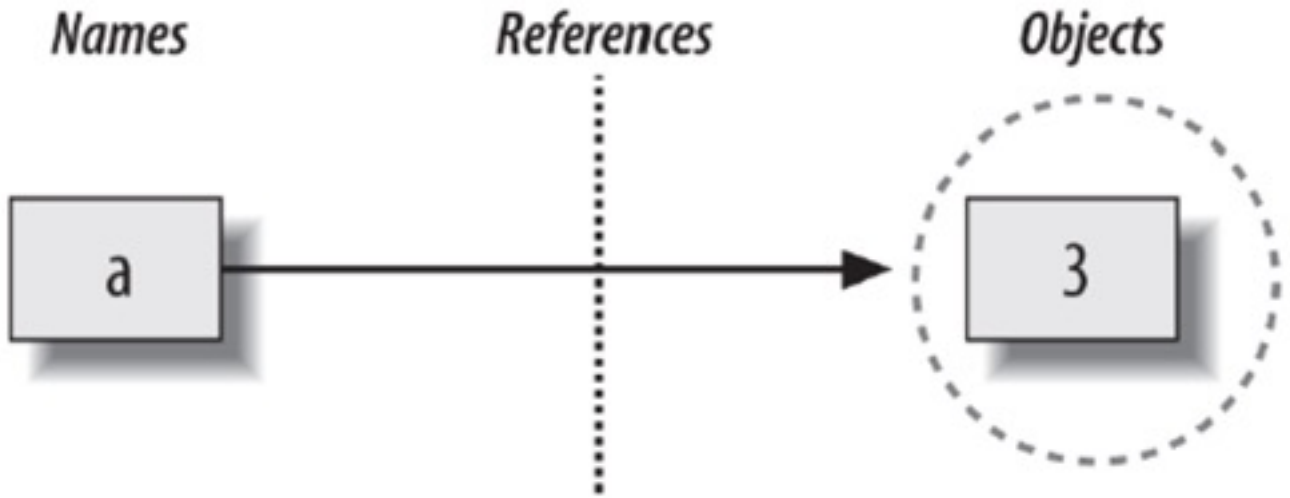
- 序列运算无效
  - 字典元素之间没有顺序的概念，类似分片（提取相邻片段）的运算是不能用的。
- 对新索引赋值会添加项
- 键不一定总是字符串
  - 此前的例子中都是用字符串作为键，但任何**不可变对象**都是可以的。例如可以用整数作为键，这样字典看起来很像列表

课程内容

- 列表
  - 列表的基本操作
  - 列表常用操作
- 字典
  - 字典的基本操作
  - 字典常用操作
- 动态类型

动态类型

- a = 3
- Python 会执行三个不同的步骤去完成这个请求：
  - 创建一个对象来代表数值3
  - 创建一个变量 a（如果它还没有被创建的话）
  - 将变量与新的对象 3 相连接。
-



类型属于对象，而不是变量

- 在 Python 中从变量到对象的连接称作**引用**。
- 引用是一种关系，以内存中的指针形式实现。一旦变量被使用，Python 自动追踪这个变量到对象的连接。
  - **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。
  - **引用**是自动形成的从变量到对象的指针。

```
a = 3                    # It's an integer
a = 'spam'               # Now it's a string
a = 1.23                 # Now it's a floating point
```

- 我们没有改变变量 a 的类型，只是把 a 修改为对不同对象的引用。另一方面，对象知道自己的类型。
- 每个对象都包含一个**头信息**，其中标记了这个对象的类型。
  - 例如，整数对象 3，包含了数值 3 以及一个头信息，用来告诉 Python 这是一个整数对象。

垃圾收集（GC）

- 我们把变量 a 赋值给了不同类型的对象，它前一个引用值发生了什么变化？

```
a = 3                         # It's an integer
a = 'spam'                    # Now it's a string
a = 1.23                      # Now it's a floating point
```

垃圾收集（GC）

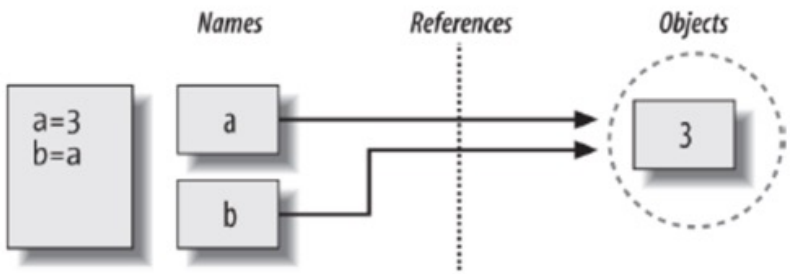
- 我们把变量 a 赋值给了不同类型的对象，它前一个引用值发生了什么变化？
  - 答案是，之前的那个对象占用的空间就会被回收（如果它没有被其它的变量名或对象所引用的话）。这种自动回收对象空间的技术叫做**垃圾收集（GC, Garbage Collection）**。
- 在 Python 内部，它在每个对象中保持了一个计数器，计数器记录了当前指向该对象的引用的数目。一旦这个计数器被设置为零，这个对象的内容空间就会被自动回收。

- 垃圾收集最直接的好处是：在脚本中任意使用对象而不需要考虑释放内存空间。在程序运行时，Python 会自动清理那些不再使用的空间。

共享引用

- 引入另一个变量,发生什么变化

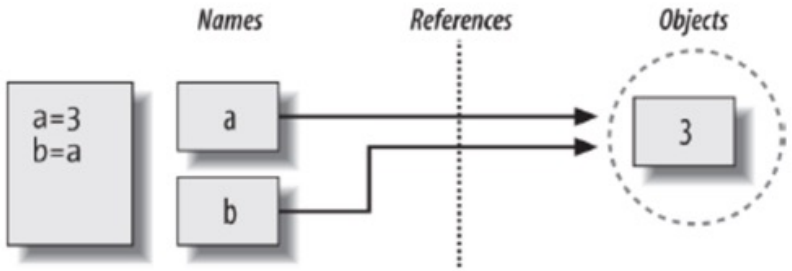
```
a = 3
b = a
```



共享引用

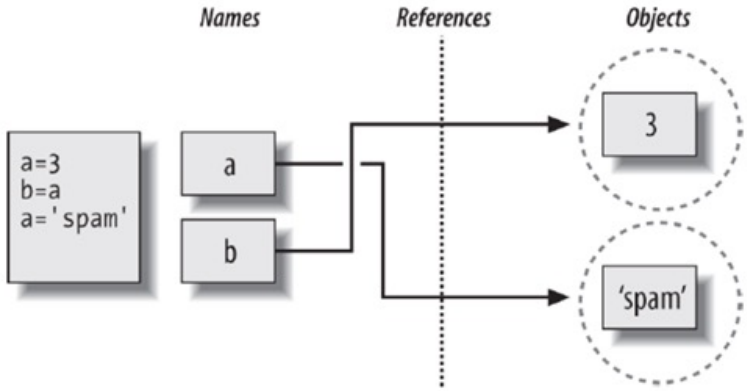
- 引入另一个变量,发生什么变化

```
a = 3
b = a
```



- 多个变量名引用了同一个对象，这在 Python 中叫做共享引用。

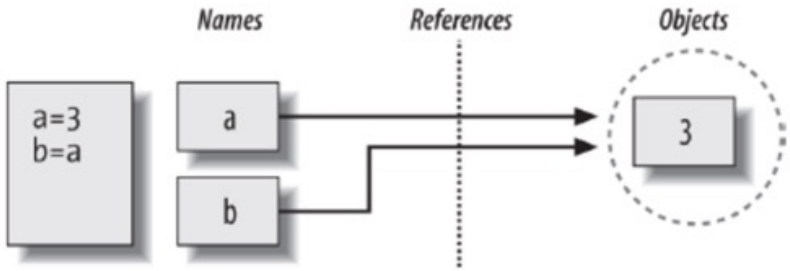
```
a = 3
b = a
a = 'spam'
```



共享引用

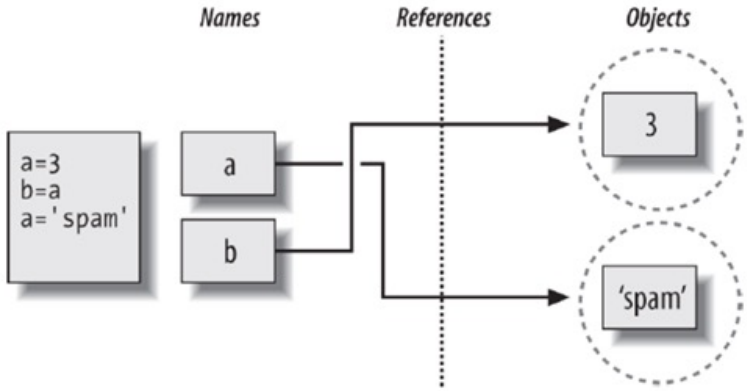
- 引入另一个变量,发生什么变化

```
a = 3
b = a
```



- 多个变量名引用了同一个对象，这在 Python 中叫做共享引用。

```
a = 3
b = a
a = 'spam'
```



共享引用

- 看下面三个语句

```
a = 3
b = a
a = a + 2
```

共享引用

- 看下面三个语句

```
a = 3
b = a
a = a + 2
print(a,b)
```

```
5 3
```

最后的赋值将 a 设置为 5 的引用，而并不会产生改变了 b 的副作用

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，并不是替换了原始对象，而是让这个变量去引用完全不同的另一个对象。

共享引用和在原处修改

```
L1 = [2, 3, 4]
L2 = L1
```

L1 是一个包含对象 2、3 和 4 的列表，所以 L1[0] 引用对象 2，它是列表 L1 中的第一个元素。
在运行两个赋值后，L1 和 L2 引用了相同的对象，就像我们之前例子中的 a 和 b 一样。
那如果我们去修改 L1 列表，会发生什么情况？

共享引用和在原处修改

```
L1 = [2, 3, 4]                  # A mutable object
L2 = L1                         # Make a reference to the same object
L1[0] = 24                      # An in-place change
print(L1)
print(L2)
```

```
[24, 3, 4]
[24, 3, 4]
```

- 改变了 L1 所引用的对象的一个元素，这个在原处的改变不仅仅会对 L1 有影响，也会对 L2 产生影响，因为 L2 与 L1 都引用了相同的对象。

共享引用和在原处修改

```
L1 = [2, 3, 4]                  # A mutable object
L2 = L1                         # Make a reference to the same object
L1[0] = 24                      # An in-place change
print(L1)
print(L2)
```

```
[24, 3, 4]
[24, 3, 4]
```

- 改变了 L1 所引用的对象的一个元素，这个在原处的改变不仅仅会对 L1 有影响，也会对 L2 产生影响，因为 L2 与 L1 都引用了相同的对象。
- 如果你不想要这样的现象发生，需要 Python 拷贝对象，而不是创建引用。

```
L1 = [2, 3, 4]
L2 = L1[:]                      # Make a copy of L1 (or list(L1), copy.copy(L1), etc.)
L1[0] = 24
print(L1)
print(L2)
```

```
[24, 3, 4]
[2, 3, 4]
```

共享引用和相等

- 由于 Python 的引用模型，在 Python 程序中有两种不同的方法去检查是否相等。
  - 第一种是 **"=="** 操作符，**测试两个被引用对象是否有相同的值**。
  - 第二种是 **"is"** 操作符，**检查对象的同一性**。如果两个变量名精确指向同一个对象，它会返回 **True**，所以是更严格形式的相等测试。

```
L = [1, 2, 3]
M = L                          # M and L reference the same object
L == M                         # Same values
```

```
True
```

```
L is M                         # Same objects
```

```
True
```

共享引用和相等

- **is** 是代码中检测共享引用的一种方法。如果变量名引用值相等，但是是不同的对象，它的返回值是 **False**

```
L = [1, 2, 3]
M = [1, 2, 3]
L == M
```

```
True
```

```
L is M
```

```
False
```

小结

- 本章探讨了列表和字典。
  - Python 程序中两种最常见、最具有灵活性、功能最强大的两种集合体类型
- 列表类型支持任意对象的以位置排序的集合体，而且可以任意嵌套，按需增长和缩短。
- 字典类型也是如此，不过它是以键来存储元素而不是位置，并且不会保持元素之间的顺序关系。
- 列表和字典都是可变的，所以它们支持各种不适用于字符串的原处修改操作。
  - 例如，列表可以通过 append() 方法来进行增长，而字典通过赋值给新键来实现增长。

练习

- 举出两种方式来创建内含五个整数零的列表。
- 创建一个字典，有26个键从"A"到"Z"，每个键关联的值是从1到26