Lecture 2　Data Structures & Control Flow

金融科技协会　2018 年 10 月 12 日

目录

一 数据结构

Python 有 6 种常用的数据结构 (Data structure，又称为“容器”，Container)，简记为“4+1+1”，分别为：

- 4 种序列类型 (Sequence Type)，即 list(列表序列)、str(字符串序列)、tuple(元组序列)、range(规则整数序列)；
- 1 种集合类型 (Set Type)，即 set(集合)；
- 1 种映射类型 (Mapping Type)，即 dict(字典)。

上述 6 种数据结构又可以根据**是否可变**分为两大类：**可变容器 (mutable)**——list、dict、set，可以修改容器中已有的数据；**不可变容器 (immuable)**：str、range、tuple，不可以修改容器中已有的数据。

```
>>> nameStr = "Draymond"
>>> nameStr[0] = 'd'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>>
>>> nameList = list(nameStr)
>>> nameList
['D', 'r', 'a', 'y', 'm', 'o', 'n', 'd']
>>> nameList[0] = 'd'
>>> nameList
['d', 'r', 'a', 'y', 'm', 'o', 'n', 'd']
```

图 1: 修改字符串对象报错，而修改列表对象则不会

1.1 列表 (list)

列表 (list) 是一个任意类型的对象的位置相关的有序集合。这里有两个关键词，第一个是**任意类型**，列表中的元素没有固定类型的约束，允许类型不相同，这也是列表和数组的最重要的区别 (数组，此数据结构中的元素类型必须相同)；第二个是**位置相关**，列表是有序的，每个元素都有自己的位置信息，也就是索引，从 0 开始。

1.1.1 列表的创建

list 的创建主要有三种方式 (图 2)：

- 第一，使用中括号直接创建；
- 第二，使用 list 函数创建；
- 第三，使用列表解析创建 (暂略)。

```python
# 使用中括号直接创建
aList = [1,2,3,'A']
# 使用list函数创建
bList = list('Draymond')
# 使用列表解析创建
cList = [str(x) * 2 for x in aList]

# 显示结果
print(aList)
print(bList)
print(cList)
```

```
[1, 2, 3, 'A']
['D', 'r', 'a', 'y', 'm', 'o', 'n', 'd']
['11', '22', '33', 'AA']
[Finished in 0.1s]
```

图 2: 列表的三种创建方式

1.1.2 序列操作

对包括列表在内的所有序列类型 (即 list, tuple, str, range) 的操作可以分为两大类：第一是**序列操作**，即适用于所有序列类型的通用操作；第二是**特有操作**，即每个序列类型自己所独有的操作，对其他序列类型不适用。因此，本节所介绍的序列操作，同样适用于元组、字符串以及规则整数序列，下文便不再赘述。

+：加号，表示合并。

∗：乘号，表示重复。

in 与 *not in*：判断元素是否在目标序列中。

len()：返回序列长度。

$X[J]$：索引，index。索引从 0 开始，最后一个是 n-1；支持反向索引，倒数第一个元素索引是 -1，倒数第二个是 -2，以此类推。

```python
indexSample = list(' Shadow ')
print(indexSample[0])          # 第1个元素，是空格
print(indexSample[1])          # 第2个元素，是S
print(indexSample[-1])         # 倒数第1个元素，是空格
print(indexSample[-2])         # 倒数第2个元素，是w

```

```
S

w
[Finished in 0.2s]
```

图 3: 列表索引

$X[I:J:Y]$：切片，slice。切片指按照相关规则截取序列的一部分。可切片序列的截取规则相同，有三个参数：

- 第一个参数表示左边界 (闭)，默认为 0，即第一个元素；
- 第二个参数右边界 (开，即不包括)，默认为 n，即默认取到索引为 n-1 的那个元素，也就是最后一个元素；
- 第三个参数是截取间隔 (步长)，默认是 1，即连续截取；步长可以为负数，代表从后往前取。

```python
sliceSample = list('ABCDEF')
print('01', sliceSample, sep='===>')          # 显示sliceSample
print('02', sliceSample[:], sep='===>')       # 切出所有元素
print('03', sliceSample[1:], sep='===>')      # 切出第2个到最后一个
print('04', sliceSample[:-1], sep='===>')     # 切出第1个到倒数第二个(不包括最后一个)
print('05', sliceSample[2:5], sep='===>')     # 切出第3个到第5个元素
print('06', sliceSample[:4:2], sep='===>')    # 隔一个切一下，即切出索引为0和2的元素
print('07', sliceSample[::-1], sep='===>')    # 切出所有元素，但是从后往前切
```

```
01===>['A', 'B', 'C', 'D', 'E', 'F']
02===>['A', 'B', 'C', 'D', 'E', 'F']
03===>['B', 'C', 'D', 'E', 'F']
04===>['A', 'B', 'C', 'D', 'E']
05===>['C', 'D', 'E']
06===>['A', 'C']
07===>['F', 'E', 'D', 'C', 'B', 'A']
[Finished in 0.1s]
```

图 4: 列表切片

1.1.3 特有操作

列表没有固定大小，可以增加或减小，常用的方法如下 (特别注意：因为列表是可变的，所以绝大多数列表方法会**就地改变**列表对象，而不是创建一个新的列表)：

- $append(value)$，在列表的尾部插入一项；
- $pop(index)$，移除指定索引处的项，没有传入参数则删除最后一项，会将删除的元素返回；
- $insert(index, value)$，~~按照值移除元素，只移走第一个 (后面还有的不移走)~~；在index处插入value
- del，也是删除，$del\ listA[1:4]$ 意思是删除 $listA$ 的索引为 1，2，3 的元素；$del\ listA$ 意思是删除 $listA$ 这个变量，即 $listA$ 不再存在；
- $clear()$，清除列表中的全部元素，使其成为空的 list。

– remove(value)：按照值移除元素，只移走第一个（后面还有的不移走）

```
>>> fruit = ['apple','pear']
>>> fruit.append('grape')
>>> fruit
['apple', 'pear', 'grape']
>>> fruit.append('watermelon')
>>> fruit
['apple', 'pear', 'grape', 'watermelon']
>>> fruit.append('mongo')
>>> fruit
['apple', 'pear', 'grape', 'watermelon', 'mongo']
>>> fruit.pop()
'mongo'
>>> fruit
['apple', 'pear', 'grape', 'watermelon']
>>> fruit.pop(0)
'apple'
>>> fruit
['pear', 'grape', 'watermelon']
>>> fruit.insert(0, 'apple')
>>> fruit
['apple', 'pear', 'grape', 'watermelon']
>>> fruit.insert(4, 'apple')
>>> fruit
['apple', 'pear', 'grape', 'watermelon', 'apple']
>>> fruit.remove('apple')
>>> fruit
['pear', 'grape', 'watermelon', 'apple']
>>> fruit.clear()
>>> fruit
[]
```

图 5: 列表方法 (一)

- $sort()$，默认升序对列表进行排序，可以指定是否反序：$sort(reverse = True)$，还有一个关键字参数 key，可以指定排序时所依赖的值；
- $reverse()$，对列表进行翻转。

注意和sorted函数区别：该函数返回一个新列表，而非就地修改

```
>>> data = [7,9,2,1,0,-8,-10,-3,2,9,1]
>>> data.sort()
>>> data
[-10, -8, -3, 0, 1, 1, 2, 2, 7, 9, 9]
>>> data.sort(reverse = True)
>>> data
[9, 9, 7, 2, 2, 1, 1, 0, -3, -8, -10]
>>>
>>>
>>> data2 = [['Bob',93], ['John',89], ['Jane', 95], ['Sam', 76]]
>>> data2.sort(key = lambda x:x[1])
>>> data2
[['Sam', 76], ['John', 89], ['Bob', 93], ['Jane', 95]]
```

图 6: 列表方法 (二)

上例中传给 key 参数的是一个 **lambda 表达式**，是函数的一种形式，$lambda\ x : x[1]$ 表示，传入参数为 x 而返回的结果是 $x[1]$ 的一个函数。上例的意思是，根据 $data2$ 中每个元素列表的第二个值的大小来排序。

1.1.4 列表嵌套

Python 会进行**边界检查**。Python 不允许引用不存在的元素，也就是超出列表末尾之外的索引。

```
>>> A = [1,2,3,4,5]
>>> A[100]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

图 7: IndexError

嵌套：Python 支持嵌套，即列表中的元素可以是列表，可以是列表中套列表，等等。此时特别注意索引的使用：

```
>>> listA = [[1,2,3], [4,5,6], [7,8,9]]
>>> listA
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> listB = [{'a':'A'}, ['b','c'], 'd']
>>> listB
[{'a': 'A'}, ['b', 'c'], 'd']
>>>
>>>
>>> listA[0][0]
1
>>> listA[0][1]
2
>>> listA[0][2]
3
>>> listB[0]['a']
'A'
```

图 8: 嵌套与索引

1.2 元组 (tuple)

元组和列表十分类似，但是元组**不能改变**——不能增加或插入新元素，不能减少已有元素。因此，元组可以简单理解为不可以改变的列表。元组使用圆括号而不是方括号，圆括号有时可以省略；由于 tuple 是序列的一种，因此序列操作同样适用于元组，如：

```
>>> T1 = (1,2,3,4,5)
>>> T2 = 1,2,3,4,5
>>> T1 == T2
True
>>> T1[0]
1
>>> T1[1:4]
(2, 3, 4)
>>> len(T)
5
>>> T + (6,7)
(1, 2, 3, 5, 6, 6, 7)
>>> T + [8,9]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate tuple (not "list") to tuple
```

图 9: 元组的创建与序列操作

1.2.1 特有操作

元组有两个专有的可调用方法：

- *index*()：返回参数的索引，若目标参数在元组中出现多次，则只返回第一个出现的目标参数的索引；若目标参数未在元组中，则会报错；
- *count*()：返回目标参数在元组中出现的次数，没有出现则返回 0。

```
>>> T = tuple('draymond')
>>> T
('d', 'r', 'a', 'y', 'm', 'o', 'n', 'd')
>>> T.index('d')
0
>>> T.count('d')
2
```

图 10: 元组的方法

1.2.2 元组的优势

有了 list，为何还要使用 tuple?

第一，不可变性——提供了一个完整性约束；

第二，如果数据组成相同，tuple 相对于 list 一般说来速度更快且占用内存空间更少。

1.2.3 其他

当声明仅有一个元素的 tuple 时，必须加上逗号以规避歧义，否则创建的不是一个 tuple 而是其他类型。

可以使用 *tuple*() 将 list 转化为 tuple。

注：元组本身不可以修改，但是如果元组的元素是列表，那么里面的列表是可以修改的。
>>> a = ([1,2], [3,4])
>>> a[0][0] = 222
>>> a
>>> ([222, 2], [3,4])
>>> a[0] = [1,2]
TypeError: 'tuple' object does not support item assignment

```
>>> T1 = (1)          # T1不是元组而是整数
>>> type(T1)
<class 'int'>
>>> T2 = ([1,2,3])    # T2不是元组而是列表
>>> type(T2)
<class 'list'>
>>>
>>> T1 = (1,)         # 此时T1是元组
>>> type(T1)
<class 'tuple'>
```

图 11: 单个元素的元组的创建

1.3 字符串 (str)

基本操作参见 Lecture 1。

1.3.1 不可变性

字符串具有不可变性，在创建后不能**就地改变**，例如：不能通过对其某一位置进行赋值而改变字符串：

```
>>> strA = 'Hello'
>>> strA[0] = 'h'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>>
```

图 12: TypeError

1.3.2 字符串的方法

注意：留意措辞，都是**“返回……”**——由于字符串是不可变对象，因此以下方法都创建了一个新的字符串，而不是改变原有的字符串。

- *upper*()：返回大写；
- *lower*()：返回小写；
- $find(A)$：子字符串查找操作，返回一个子字符串的偏移量 (索引)，没有的话返回 -1，找到第一个就停止 (不管后面有没有了)；
- $replace(A, B)$：全局进行搜索替换，两个参数，用第二个参数替换第一个参数，默认替换所有；可增加第三个参数 *count*，1 代表只替换第一个，2 代表只替换前两个，以此类推；
- $split(A)$：按照给定的分隔符 A 切分字符串，返回一个 list；可以设置第二个参数 $maxsplit$，表示只按前 $maxsplit$ 个分隔符切割；
- *join*()：字符串穿插，$strA.join(strB)$，将 $strA$ 穿插在 $strB$ 的每个字符之间；
- *strip*()：删除字符串两端空格并返回；
- *lstrip*()：删除字符串左侧空格并返回；

```
>>> 'apple'.upper()
'APPLE'
>>> 'PEAR'.lower()
'pear'
>>> '123aaa123bbb123'.find('123')
0
>>> '123aaa123bbb123'.replace('123', '456')
'456aaa456bbb456'
>>> '123aaa123bbb123'.replace('123', '456', 1)
'456aaa123bbb123'
>>> '123aaa123bbb123'.replace('123', '456', 2)
'456aaa456bbb123'
>>> '123aaa123bbb123'.split('123')
['', 'aaa', 'bbb', '']
>>> '123aaa123bbb123'.split('123', 1)
['', 'aaa123bbb123']
>>> '123aaa123bbb123'.split('123', 2)
['', 'aaa', 'bbb123']
>>> '='.join('ABCDE')
'A=B=C=D=E'
>>> '    abc    '.strip()
'abc'
>>> '    abc    '.lstrip()
'abc    '
>>> '    abc    '.rstrip()
'    abc'
```

图 13: 字符串常用方法

- *rstrip*()：删除字符串右侧空格并返回。

再次提醒，上面的所有方法，不会改变原始字符串，而是会创建一个新的字符串——字符串是不可变对象，必须这样。

```
>>> A = [1,2,3,4]
>>> str(A)
'[1, 2, 3, 4]'
>>>
>>> ord('A')
65
>>> chr(65)
'A'
>>>
```

图 14: 三个内置函数

三个特殊的内置函数：

- *str*()：从其他类型对象获得字符串对象；
- *ord*()：返回目标字符的序号；
- *chr*()：返回序号对应的字符，是 order 的逆操作。

1.3.3 格式化操作

字符串的格式化操作可以使代码书写更加简洁直观，有两种形式：

- 表达式形式：$'\%s,\ pear,\ and\ \%s' \ \%\ ('apple',\ 'watermelon')$

- 字符串方法形式：$'\{0\},\ eggs,\ and\ \{1\}'.format('apple',\ 'watermelon')$

```
>>> 'I like %s and %s.' % ('apple', 'pear')
'I like apple and pear.'
>>> 'I like {0} and {1}.'.format('apple', 'pear')
'I like apple and pear.'
>>>
```

图 15: 格式化操作

1.4 规则整数序列 (range)

见 for 循环部分。

1.5 集合 (set)

set 不是序列 (Sequence Type)，而是 Set Types 之一。

python 中的 set 类型指“集合”，和数学中的集合概念相同，用于存储无重复值，且无序。可以使用大括号或 *set*() 函数创建集合：

```
>>> aFruit = {'apple', 'mongo', 'banana', 'cherry'}
>>> bFruit = set(['mongo', 'pear', 'cherry'])
>>> aFruit
{'apple', 'mongo', 'banana', 'cherry'}
>>> bFruit
{'mongo', 'cherry', 'pear'}
>>> type(aFruit)
<class 'set'>
>>> type(bFruit)
<class 'set'>
>>> len(aFruit)
4
>>> len(bFruit)
3
```

图 16: 集合创建

若创建时，给予集合重复元素，则创建后自动删除。因此集合也经常用于去重工作：

```
>>> hasRepeat = ['apple', 'apple', 'pear']
>>> set(hasRepeat)
{'apple', 'pear'}
>>> {'apple', 'apple', 'apple'}
{'apple'}
>>>
```

图 17: 元素不重复 & 去重

使用 *add*() 和 *remove*() 方法对 Set 类型增加或者删除成员。

```
>>> aFruit
{'apple', 'mongo', 'banana', 'cherry'}
>>> bFruit
{'mongo', 'cherry', 'pear'}
>>> aFruit.add('watermelon')
>>> aFruit
{'apple', 'watermelon', 'banana', 'mongo', 'cherry'}
>>> aFruit.remove('watermelon')
>>> aFruit
{'apple', 'banana', 'mongo', 'cherry'}
>>>
```

图 18: add & remove

使用运算符进行集合运算：

- &：交集
- |：并集
- −：差集
- ^：对称差集

```
>>> aFruit
{'apple', 'banana', 'mongo', 'cherry'}
>>> bFruit
{'mongo', 'cherry', 'pear'}
>>> aFruit & bFruit
{'mongo', 'cherry'}
>>> aFruit | bFruit
{'banana', 'apple', 'mongo', 'cherry', 'pear'}
>>> aFruit - bFruit
{'apple', 'banana'}
>>> bFruit - aFruit
{'pear'}
>>> aFruit ^ bFruit
{'apple', 'banana', 'pear'}
```

图 19: 集合运算 (运算符)

使用方法进行集合运算：

- $intersection()$：交集
- $union()$：并集
- $difference()$：差集
- $symmetric\_difference()$：对称差集

```
>>> aFruit
{'apple', 'banana', 'mongo', 'cherry'}
>>> bFruit
{'mongo', 'cherry', 'pear'}
>>> aFruit.intersection(bFruit)
{'mongo', 'cherry'}
>>> aFruit.union(bFruit)
{'banana', 'apple', 'mongo', 'cherry', 'pear'}
>>> aFruit.difference(bFruit)
{'apple', 'banana'}
>>> bFruit.difference(aFruit)
{'pear'}
>>> aFruit.symmetric_difference(bFruit)
{'apple', 'banana', 'pear'}
```

图 20: 集合运算 (方法)

1.6 字典 (dict)

dict 也非序列，而是 Mapping Types(映射) 之一。映射不是通过位置 (索引) 来存储数据，而是通过**键值对 (key-value)**存储。也就是说，映射没有从左到右的顺序。字典具有可变性，可以随需求增大或减小，类似于列表。

字典由一系列键值对构成，这些键值对就代表着映射。例如下面的字典，含有三个元素:

$$myDog = \{'name' :' Sam', 'age' : 5, 'color' :' white'\}$$

其中，字符串 name、age 和 color 是键 (key)，字符串 Sam、white 和数字 5 是值，对应关系是 name 对 Sam，age 对 5，color 对 white。

注意：字典中键 (key) 不可以重复，但值 (value) 可以重复。

1.6.1 字典的创建

```
1  # 直接创建
2  A = {'Bob':'apple', 'Jane':'pear'}
3  print(A)
4  # 从零添加
5  B = {}
6  B['Bob'] = 'apple'
7  B['Jane'] = 'pear'
8  print(B)
9  # dict函数
10 C = [['Bob', 'apple'], ['Jane', 'pear']]
11 C = dict(C)
12 print(C)
13
{'Bob': 'apple', 'Jane': 'pear'}
{'Bob': 'apple', 'Jane': 'pear'}
{'Bob': 'apple', 'Jane': 'pear'}
[Finished in 0.3s]
```

图 21: 创建字典

- 直接创建：键值对用冒号连接，写在花括号里；
- 从零添加：先创建一个空字典，然后添加键值对；

- dict 函数：向 dict 函数中传入列表 (元组也可以)；或者直接 dict() 创建一个空字典，然后再添加。

1.6.2 获取值

可以通过键 (key) 来访问值 (value)：

```
>>> myDog = {'name':'Sam', 'age':5, 'color':'white'}
>>> myDog['name']
'Sam'
>>> myDog['age']
5
>>> myDog['color']
'white'
```

图 22: 访问值

也可以通过键 (key) 来改变值 (value)：

```
>>> myDog
{'name': 'Sam', 'age': 5, 'color': 'white'}
>>> myDog['name'] = 'Bob'
>>> myDog
{'name': 'Bob', 'age': 5, 'color': 'white'}
>>> myDog['age'] = 2
>>> myDog
{'name': 'Bob', 'age': 2, 'color': 'white'}
>>> myDog['color'] = 'black'
>>> myDog
{'name': 'Bob', 'age': 2, 'color': 'black'}
```

图 23: 修改值

与列表对比一下：方括号里，字典是键，列表是相对位置 (数字)。

注意：给一个不存在的键赋值不会产生错误，但访问一个不存在的键会导致错误。

```
>>> X
{'d': 9, 'a': 3, 'c': 0, 'b': 7}
>>> X['e']
Traceback (most recent call last):
  File "<pyshell#201>", line 1, in <module>
    X['e']
KeyError: 'e'
```

图 24: KeyError

可以使用 in 关系表达式来进行测试 (图 25)。

```
>>> X
{'d': 9, 'a': 3, 'c': 0, 'b': 7}
>>> 'a' in X
True
>>> 'e' in X
False
>>> 'e' not in X
True
>>> if 'e' not in X:
        print('missing')


missing
```

图 25: in & not in

get 方法也可以用来获取值，是带有一个默认值的访问方法，如下 (图 26)：

```
>>> X = {'d':9, 'a':3, 'c':0, 'b':7}
>>> X.get('d')
9
>>> X.get('e')
>>> X.get('F')
>>>
```

图 26: get 方法

默认情况 (不设置第二个参数，默认为 None)，如果 get 的参数在字典的键中有，则返回相应的值；若没有，则什么都不返回，也不报错。

第二个参数，意思是：若字典中没有目标键，则返回该值，可以默认为 None，也可以自己设置：

```
>>> X.get('e', 10)
10
>>> X.get('f', 'do not exist')
'do not exist'
>>>
```

图 27: get 方法设置

上例中，字典中没有键 e，则返回了 10；字典中没有键 g，则返回了“do not exist”。

1.6.3 嵌套

字典支持嵌套：还是以键为中心，通过键 (key) 来层层访问或修改值 (value)：

```
>>> myCat
{'name': 'Jane', 'age': 2, 'color': 'brown'}
>>> myCat['favorite'] = {'food':'fish', 'toy':'ball', 'sport':'sleeping'}
>>> myCat
{'name': 'Jane', 'age': 2, 'color': 'brown', 'favorite': {'food': 'fish', 'toy':
 'ball', 'sport': 'sleeping'}}
>>> myCat['favorite']['food']
'fish'
```

图 28: 嵌套

1.6.4 顺序

字典中的元素 (键值对) 是没有顺序的，也就是说，我们建立一个字典并将其打印出来，它的键值对的顺序也许与输入时的顺序不同。

如果实在是需要以某种顺序调取某个字典的所有值 (value)，那么需要：

- 用 *.keys* 方法收集一个字典的键的列表 (需要用 *list*() 转化)；
- 对键的列表按需求排序；
- 用 for 循环。

下例：

```
>>> X = {'d':9, 'a':3, 'c':0, 'b':7}
>>> X
{'d': 9, 'a': 3, 'c': 0, 'b': 7}
>>> keys = list(X.keys())
>>> keys
['d', 'a', 'c', 'b']
>>> keys.sort()
>>> keys
['a', 'b', 'c', 'd']
>>> for key in keys:
        print(X[key])


3
7
0
9
```

图 29: 顺序访问

1.6.5 字典方法

除了上面已经用到的 *get*() 方法，字典还有 *keys*()、*values*()、*items*() 等方法。

- *keys*：如图，keys 方法返回一个包含字典中所有键 (key) 的特殊的对象，类似于 list 但不是 list，它其实是一个可迭代对象，可以使用 for 循环；可以使用 list 函数将其转化成 list。

```
>>> myDog
{'name': 'Sam', 'age': 5, 'color': 'white'}
>>> keys = myDog.keys()
>>> keys
dict_keys(['name', 'age', 'color'])
>>> type(keys)
<class 'dict_keys'>
>>> keys = list(keys)
>>> keys
['name', 'age', 'color']
>>>
>>> values = myDog.values()
>>> values
dict_values(['Sam', 5, 'white'])
>>> type(values)
<class 'dict_values'>
>>> values = list(values)
>>> values
['Sam', 5, 'white']
>>>
>>> items = myDog.items()
>>> items
dict_items([('name', 'Sam'), ('age', 5), ('color', 'white')])
>>> type(items)
<class 'dict_items'>
>>> items = list(items)
>>> items
[('name', 'Sam'), ('age', 5), ('color', 'white')]
```

图 30: 字典方法

- *values*：如图，values 方法返回一个包含字典中所有值 (value) 的特殊的对象，类似于 list 但不是 list，它其实是一个可迭代对象，可以使用 for 循环；可以使用 list 函数将其转化成 ilst。
- *items*：如图，items 方法返回一个包含字典中所有键值对 (key-value，也叫 item) 的特殊的对象，类似于 list 但不是 list，它实是一个可迭代对象，可以使用 for 循环；可以使用 list 函数将其转化成 list，里面的元素是元组。

1.7 小结

```
>>> dir([1,2,3])
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__
    ', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '
    __hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '
    __le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '
    __reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__',
    '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', '
    extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
>>> dir({'a':'A', 'b':'B'})
['__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__'
    , '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '
    __init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__ne__', '
    __new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__setitem__', '
    __sizeof__', '__str__', '__subclasshook__', 'clear', 'copy', 'fromkeys', 'get', 'items
    ', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']
>>>
```

图 31: *dir*()

忘了序列有哪些属性或方法可供调用？使用内置函数 *dir*()，该方法将返回一个列表，包含目标对象的所有属性和方法：

另外，*dir*() 函数只是简单给出了方法的名称，而 Lecture 1 中介绍的 *help*() 函数能够具体查询一个属性或方法是干什么的，两者经常配合使用。

二 选择

2.1 布尔表达式

布尔表达式是能计算出一个布尔值 *True* 或 *False* 的表达式，具体见 Lecture 1 中“比较运算符”与“逻辑运算符”部分。

2.2 if 语句

如果条件为真，就执行某一语句，如下：

boolean expression
不需要加括号

```
if boolean-expression:
    statement(s)
```

图 32: if 语句结构

上述语句的含义是：如果 if 后的布尔表达式 (即条件) 为真，则执行其中的 statement(s) 语句，否则不执行，直接跳过。

下例中，如果 age 大于 25，则 age 的值加上 3；因为 age 是 30，所以 $age > 25$ 这个布尔表达式的结果是 *True*，所以执行 if 块中的语句，因此最后 age 为 33。

```
1 age = 30
2 if age > 25:
3     age += 3
4 print(age)

33
[Finished in 0.2s]
```

图 33: age 最终为 33

注意，if 块中的语句要缩进，且要以相同的空白缩进，相同的方式缩进 (不能一行是一个 tab 缩进，一行是 4 个空格缩进，虽然他们的视觉效果是一样的)。下面的例子与上面的例子虽然只差了一个缩进，但含义完全不一样。下例中，*print*(*age*) 被包含在 if 块内，如果条件不满足，就不会执行 *print* 语句：

```
1 age = 22
2 if age > 25:
3     age += 3
4     print(age)

[Finished in 0.3s]
```

图 34: 缩进不同导致差异

2.3 if-else 语句

if-else 语句的结构如下：

```
if boolean-expression:
    statement(s)-for-the-true-case
else:
    statement(s)-for-the-false-case
```

图 35: if-else 语句结构

上述语句的含义是：如果 if 后的布尔表达式 (即条件) 为真，则执行 if 块中的 statement(s) 语句，否则执行 else 块中的语句。

下例中，如果 age 大于 25，则会打印“old”；否则 (小于等于 25)，则会打印“young”；由于 age 为 30，故将打印“old”：

```
1 age = 30
2 if age > 25:
3     print("old")
4 else:
5     print("young")

old
[Finished in 0.3s]
```

图 36: 打印 young

2.4 嵌套 if 语句与 elif 的使用

将一个 if 语句放在另一个 if 语句中就形成了一个嵌套 if 语句，如下例：

```
score = 75
if score >= 90:
    grade = 'A'
else:
    if score >= 80:
        grade = 'B'
    else:
        if score >= 70:
            grade = 'C'
        else:
            grade = 'D'
print(grade)
```

```
C
[Finished in 0.3s]
```

图 37: 嵌套 if 语句

含义：如果 score 大于等于 90，grade 为 A，否则继续判断，如果 score 大于等于 80，grade 为 B，否则继续判断，如果 score 大于等于 70，grade 为 C，否则为 D。

使用 *elif* 可以简写嵌套 if 语句，*elif* 表示“否则如果”，下例中的语句和上例完全等价：

```
score = 75
if score >= 90:
    grade = 'A'
elif score >= 80:
    grade = 'B'
elif score >= 70:
    grade = 'C'
else:
    grade = 'D'
print(grade)
```

```
C
[Finished in 0.4s]
```

图 38: elif 的使用

2.5 if-else 三元表达式

```
if X:
    A = Y
else:
    A = Z
```

图 39: 使用 if-else 赋值

等价于：

```
A = Y if X else Z
```

图 40: if-else 三元表达式

三 循环

3.1 while 循环

while 循环：只要顶端测试一直计算到真值，就会重复执行一个语句块，例如：

```
1  count = 0
2  while count <= 5:
3      print('I am fine!')
4      count += 1
I am fine!
I am fine!
I am fine!
I am fine!
I am fine!
I am fine!
[Finished in 0.3s]
```

图 41: while 循环

while 后冒号前的部分 (此例中是 *count* <= 5) 称为循环继续条件，即其为真，循环继续运行；冒号后缩进的语句块，称为循环体，是循环继续条件为真时程序将要运行的语句块。

此例中，如果 count 小于等于 5，循环一直运行，即打印 I am fine 语句，并给 count 加 1；在循环 6 次后，count 加到了 6，此时回到循环继续条件，该条件表达式返回变成 False，循环中止，不再执行循环体中的语句。

3.2 for 循环

3.2.1 语法

python 的 for 循环通过一个序列中的每个值来进行迭代。严格来讲，for 循环是 python 中一个通用的序列迭代语句，可以遍历任何有序的序列对象内的元素，形式如下：

```
for <target> in <object>:
    <loop body>
```

图 42: for 循环结构

例如：

```
1 nameList = ['Sam', 'Jane', 'Bill']
2 for name in nameList:
3     print(name)
4 print('循环已退出')
5 print(name)
```

```
Sam
Jane
Bill
循环已退出
Bill
[Finished in 0.3s]
```

图 43: for 循环举例

第一次循环，name 变量被赋予'Sam'；第二次循环，name 变量被赋予'Jane'；第三次循环，name 变量被赋予'Bill'；所以在循环结束后，name 的值是 Bill。

3.2.2 元组赋值

```
1 T = [(1,2), (3,4), (5,6)]
2 for a,b in T:
3     print(a,b)
```

```
1 2
3 4
5 6
[Finished in 0.3s]
```

图 44: 元组赋值

上述例子可以理解为：先将第一个元素，即元组 (1,2)，赋给 (a,b)，然后根据对应关系，把 1 赋给 a，把 2 赋给 b，再执行循环体，执行一次后再进行第二轮迭代，以此类推。

元组赋值可以看做 python 在按照数据的结构进行自动解包，下例：

```
1 B = [([1,2], 3), ['XY', 6]]
2
3 for ((a,b), c) in B:
4     print(a, b, c)
```

```
1 2 3
X Y 6
[Finished in 0.2s]
```

图 45: 自动解包

3.2.3 遍历字典

用键遍历：

```
1  D = {'a':1, 'b':2, 'c':3}
2
3  for key in D:
4      print(key, D[key], sep='==>')
```

```
a==>1
b==>2
c==>3
[Finished in 0.3s]
```

图 46: 用键遍历

用元组赋值法：

```
1  D = {'a':1, 'b':2, 'c':3}
2
3  for key,value in D.items():
4      print(key, value, sep='==>')
```

```
a==>1
b==>2
c==>3
[Finished in 0.2s]
```

图 47: 用元组赋值法

3.3 range 对象

range 是常用的序列类型之一，用于产生规则变化的整数序列。range 的格式为：$range(start, end, step)$，end 的值不包含，step(步长) 默认为 1：

- $rangeA = range(5, 10)$ 相当于 $[5, 6, 7, 8, 9]$；
- $rangeB = range(10)$ 相当于 $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$
- $rangeC = range(5, 10, 2)$ 相当于 $[5, 7, 9]$
- $rangeD = range(0, -10, -2)$ 相当于 $[-8, -6, -4, -2, 0]$
- $rangeE = range(0, -10)$ 相当于 []，因为这里 end 小于 start
- $rangeD = range(0, -10, -2)$ 相当于 $[-8, -6, -4, -2, 0]$
- range 也可以切片，$range(10)[3:6]$ 结果是 $range(3, 6)$

上面的措辞是“相当于”，是因为直接打印 range 对象并不会出现这些数字，必须使用 for 循环逐一顺序访问，或者使用 list 等其他函数转化后进行打印：

```
>>> print(range(2,7))
range(2, 7)
>>> for i in range(2,7):
        print(i)
... ...
2
3
4
5
6
>>> list(range(2,7))
[2, 3, 4, 5, 6]
>>> tuple(range(2,7))
(2, 3, 4, 5, 6)
```

图 48: range 对象

3.4 break 与 continue

在循环中可以使用关键字 break 来**立即终止循环**；也可以使用关键字 continue 来结束当前迭代，控制程序转到循环体最后，即**退出一次迭代** (而 break 是退出整个循环)。

```
1 for i in range(1,10):
2     if i == 5:
3         break
4     print(i, end = ' ')

1 2 3 4 [Finished in 0.4s]
```

图 49: break 的使用

上述循环执行后输出：1 2 3 4，当 i 为 5 时，执行 if 语句块，break 会终止 for 循环。

```
1 for i in range(1,10):
2     if i == 5:
3         continue
4     print(i, end = ' ')

1 2 3 4 6 7 8 9 [Finished in 0.3s]
```

图 50: continue 的使用

上述循环执行后输出：1 2 3 4 6 7 8 9，缺 5：当 i 为 5 时，执行 if 语句块，continue 会终止当前一轮迭代，使代码直接跳过本轮迭代，即跳过 print 函数。

如果有**嵌套循环**，break 和 continue 只作用于包含该关键字的**最内层循环**。

3.5 列表解析

列表解析用于创建一个列表，其优势是代码简洁，执行速度快。其含义是：将 for 循环写在一行，将 for-in 后的容器中的每个元素拿出 (如有 if，则拿出满足 if 条件的元素)，赋给 for

后 in 前的循环变量，并执行 for 前的表达式，生成新的值，添加到新的列表中。列表解析适用于 str、list、dict、set 等支持 for 循环的容器类型。

```
1  age = [12, 23, 45, 56]
2  age2 = [i+3 for i in age]
3  print(age2)
4
5  name = 'ABCDE'
6  name2 = [n*2 for n in name]
7  print(name2)
```

```
[15, 26, 48, 59]
['AA', 'BB', 'CC', 'DD', 'EE']
[Finished in 0.3s]
```

图 51: 基本列表解析

python 的列表解析支持 if 语句，对 for 循环的成员进行筛选；也支持多个 for 循环；也可以将两者结合，即含有 if 语句的多个 for 循环的列表解析，如下：

```
1  listA = [x+10 for x in range(10) if x%3==0]
2  print(listA)
3  # listA is [10, 13, 16, 19]
4  listB=[1,2]
5  listC=[11,12,13]
6
7  # 对listB中的每个元素，和listC中的每个元素，分别求和
8  listD = [x+y for x in listB for y in listC]
9  print(listD)
10
11 # 先分别过if语句进行筛选，再分别配对求乘积
12 listE = [x*y for x in range(1,10) if x%3==0 for y in range(10,20) if y%5==0]
13 print(listE)
14
```

```
[10, 13, 16, 19]
[12, 13, 14, 13, 14, 15]
[30, 45, 60, 90, 90, 135]
[Finished in 0.2s]
```

图 52: 高阶列表解析

3.6 zip 函数

python 的内置函数 *zip*()：for 循环并行使用多个序列——以一个或多个序列为参数，然后返回元组的列表，将这些序列中的并排的元素配成对。

zip 对象和 range 对象一样，不可以直接打印 (实质是一个可迭代对象)，需要使用 list 等函数查看内容，或者使用 for 循环遍历。

```
1  L1 = [1,2,3]
2  L2 = ['a', 'b', 'c']
3
4  A = zip(L1, L2)
5
6  print(A)
7  print(list(A))
```

```
<zip object at 0x00000233B95DA308>
[(1, 'a'), (2, 'b'), (3, 'c')]
[Finished in 0.3s]
```

图 53: *zip*() 函数

zip 可以接受任何类型的序列，可以有两个以上的参数，而且可以接受不同长度的参数（zip 会以最短序列的长度为准来截断所得到的元组）：

```
1  L1 = [1,2,3]
2  L2 = ['a', 'b', 'c']
3  L3 = [1.2, 4.8, 5.3]
4  A = zip(L1, L2, L3)
5  print(list(A))
6
7  L4 = [7,8,9]
8  L5 = [0,1]
9  B = zip(L4, L5)
10 print(list(B))
```

```
[(1, 'a', 1.2), (2, 'b', 4.8), (3, 'c', 5.3)]
[(7, 0), (8, 1)]
[Finished in 0.3s]
```

图 54: 多参数 *zip*()

3.7 关于修改列表中的元素

最好不要使用 for 循环修改列表：

```
1  L1 = [1,2]
2  for x in L1:
3      x += 1
4  print(L1)
5
6  L2 = [[1,2], [3,4]]
7  for y in L2:
8      y += [1000]
9  print(L2)
10
```

```
[1, 2]
[[1, 2, 1000], [3, 4, 1000]]
[Finished in 0.3s]
```

图 55: 修改列表

为什么 L2 在修改后发生了变化，L1 却没有发生变化？

immuable object，不可变对象，如数值类型、字符串、元组等，遵循**类似 passing by value**的原则，**相当于**新拷贝一份做其他用途，新拷贝和原对象间没有联系，altering an object

inside a function will not change the original object outside.

mutable object，可变对象，如列表、字典等，遵循**类似 passing by reference**的原则，altering an object inside a function will also change the original object outside.

由于数值是 immuable object，因此在修改 L1 时，可以理解为赋给循环变量 x 的仅仅是 L1 中元素的“值”，而不是 L1 中元素“本身”(引用)，所以修改 x，对 L1 内部的元素没有任何影响。

而 L2 中的元素是列表，列表是 mutable object，所以赋给循环变量 y 的，不仅仅是“值”，而是 L2 中元素“本身”(引用)，因此对循环变量进行操作，实质上也是对 L2 中的元素进行操作，所以修改 y，对 L2 也造成了影响。

因此，为了避免上述混淆，若一定要使用循环修改列表，建议使用 range 函数，配合索引，进行操作：

```
1  L = [1,2,3,4,5,6]
2  for i in range(len(L)):
3      L[i] += 1
4
5  print(L)
```

```
[2, 3, 4, 5, 6, 7]
[Finished in 0.2s]
```

图 56: 使用 $range(len())$

或使用列表解析：$[x+1\ for\ x\ in\ L]$

By Draymond