5. 数据结构

本章节将详细介绍一些您已经了解的内容，并添加了一些新内容。

5.1. 列表的更多特性

列表数据类型还有很多的方法。这里是列表对象方法的清单：

list.append(*x*)
在列表的末尾添加一个元素。相当于 a[len(a):] = [x] 。

list.extend(*iterable*)
使用可迭代对象中的所有元素来扩展列表。相当于 a[len(a):] = iterable 。

list.insert(*i*, *x*)
在给定的位置插入一个元素。第一个参数是要插入的元素的索引，所以 a.insert(0, x) 插入列表头部，a.insert(len(a), x) 等同于 a.append(x) 。

list.remove(*x*)
移除列表中第一个值为 *x* 的元素。如果没有这样的元素，则抛出 ValueError 异常。

list.pop([*i*])
删除列表中给定位置的元素并返回它。如果没有给定位置，a.pop() 将会删除并返回列表中的最后一个元素。（ 方法签名中 *i* 两边的方括号表示这个参数是可选的，而不是要你输入方括号。你会在 Python 参考库中经常看到这种表示方法)。

list.clear()
移除列表中的所有元素。等价于``del a[:]``

list.index(*x*[, *start*[, *end*]])
返回列表中第一个值为 *x* 的元素的从零开始的索引。如果没有这样的元素将会抛出 ValueError 异常。

可选参数 *start* 和 *end* 是切片符号，用于将搜索限制为列表的特定子序列。返回的索引是相对于整个序列的开始计算的，而不是 *start* 参数。

list.count(*x*)
返回元素 *x* 在列表中出现的次数。

list.sort(*, *key=None*, *reverse=False*)
对列表中的元素进行排序（参数可用于自定义排序，解释请参见 sorted() ）。

list.reverse()
翻转列表中的元素。

list.copy()
返回列表的一个浅拷贝，等价于 a[:]。

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.count('apple')
2
>>> fruits.count('tangerine')
0
>>> fruits.index('banana')
3
>>> fruits.index('banana', 4)  # Find next banana starting a position 4
6
>>> fruits.reverse()
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
>>> fruits.append('grape')
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']
>>> fruits.sort()
>>> fruits
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
>>> fruits.pop()
'pear'
```

你可能已经注意到，像 `insert` ，`remove` 或者 `sort` 方法，只修改列表，没有打印出返回值——它们返回默认值 `None` 。[1] 这是Python中所有可变数据结构的设计原则。

你可能会注意到的另一件事是并非所有数据或可以排序或比较。 例如，`[None, 'hello', 10]` 就不可排序，因为整数不能与字符串比较，而 *None* 不能与其他类型比较。 并且还存在一些没有定义顺序关系的类型。 例如，`3+4j < 5+7j` 就不是一个合法的比较。

5.1.1. 列表作为栈使用

列表方法使得列表作为堆栈非常容易，最后一个插入，最先取出（“后进先出”）。要添加一个元素到堆栈的顶端，使用 `append()` 。要从堆栈顶部取出一个元素，使用 `pop()` ，不用指定索引。例如

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

5.1.2. 列表作为队列使用

列表也可以用作队列，其中先添加的元素被最先取出 (“先进先出”)；然而列表用作这个目的相当低效。因为在列

若要实现一个队列，可使用 collections.deque，它被设计成可以快速地从两端添加或弹出元素。例如

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.popleft()                 # The first to arrive now leaves
'Eric'
>>> queue.popleft()                 # The second to arrive now leaves
'John'
>>> queue                           # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

5.1.3. 列表推导式

列表推导式提供了一个更简单的创建列表的方法。常见的用法是把某种操作应用于序列或可迭代对象的每个元素上，然后使用其结果来创建列表，或者通过满足某些特定条件元素来创建子序列。

例如，假设我们想创建一个平方列表，像这样

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

注意，这将创建（或覆盖）一个名为 x 的变量，该变量在循环结束后仍然存在。 下述方法可以无副作用地计算平方列表:

```
squares = list(map(lambda x: x**2, range(10)))
```

或者，等价于

```
squares = [x**2 for x in range(10)]
```

上面这种写法更加简洁易读。

列表推导式的结构是由一对方括号所包含的以下内容：一个表达式，后面跟一个 for 子句，然后是零个或多个 for 或 if 子句。 其结果将是一个新列表，由对表达式依据后面的 for 和 if 子句的内容进行求值计算而得出。举例来说，以下列表推导式会将两个列表中不相等的元素组合起来:

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

而它等价于

```
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
```

```
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

注意在上面两个代码片段中， for 和 if 的顺序是相同的。

如果表达式是一个元组（例如上面的 (x, y)），那么就必须加上括号

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # create a new list with the values doubled
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filter the list to exclude negative numbers
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # apply a function to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each element
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an error is raised
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1, in <module>
    [x, x**2 for x in range(6)]
               ^
SyntaxError: invalid syntax
>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

列表推导式可以使用复杂的表达式和嵌套函数

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

5.1.4. 嵌套的列表推导式

列表推导式中的初始表达式可以是任何表达式，包括另一个列表推导式。

考虑下面这个 3x4的矩阵，它由3个长度为4的列表组成

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

如上节所示，嵌套的列表推导式是基于跟随其后的 for 进行求值的，所以这个例子等价于:

```
>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

反过来说，也等价于

```
>>> transposed = []
>>> for i in range(4):
...     # the following 3 lines implement the nested listcomp
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

实际应用中，你应该会更喜欢使用内置函数去组成复杂的流程语句。 zip() 函数将会很好地处理这种情况

```
>>> list(zip(*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

关于本行中星号的详细说明，参见 解包参数列表。

5.2. del 语句

有一种方式可以从列表按照给定的索引而不是值来移除一个元素: 那就是 del 语句。 它不同于会返回一个值的 pop() 方法。 del 语句也可以用来从列表中移除切片或者清空整个列表（我们之前用过的方式是将一个空列表赋值给指定的切片）。 例如:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

del 也可以删除整个变量

此后再引用 a 时会报错（直到另一个值被赋给它）。我们会在后面了解到 del 的其他用法。

5.3. 元组和序列

我们看到列表和字符串有很多共同特性，例如索引和切片操作。他们是 *序列* 数据类型（参见 序列类型 --- list, tuple, range）中的两种。随着 Python 语言的发展，其他的序列类型也会被加入其中。这里介绍另一种标准序列类型: *元组*。

一个元组由几个被逗号隔开的值组成，例如

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

如你所见，元组在输出时总是被圆括号包围的，以便正确表示嵌套元组。输入时圆括号可有可无，不过经常会是必须的（如果这个元组是一个更大的表达式的一部分）。给元组中的一个单独的元素赋值是不允许的，当然你可以创建包含可变对象的元组，例如列表。

虽然元组可能看起来与列表很像，但它们通常是在不同的场景被使用，并且有着不同的用途。元组是 immutable，其序列通常包含不同种类的元素，并且通过解包（这一节下面会解释）或者索引来访问（如果是 namedtuples 的话甚至还可以通过属性访问）。列表是 mutable，并且列表中的元素一般是同种类型的，并且通过迭代访问。

一个特殊的问题是构造包含0个或1个元素的元组：为了适应这种情况，语法有一些额外的改变。空元组可以直接被一对空圆括号创建，含有一个元素的元组可以通过在这个元素后添加一个逗号来构建（圆括号里只有一个值的话不够明确）。丑陋，但是有效。例如

```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

```
>>> x, y, z = t
```

这被称为 *序列解包* 也是很恰当的，因为解包操作的等号右侧可以是任何序列。序列解包要求等号左侧的变量数与右侧序列里所含的元素数相同。注意多重赋值其实也只是元组打包和序列解包的组合。

5.4. 集合

Python也包含有 *集合* 类型。集合是由不重复元素组成的无序的集。它的基本用法包括成员检测和消除重复元素。集合对象也支持像 联合，交集，差集，对称差分等数学运算。

花括号或 `set()` 函数可以用来创建集合。注意：要创建一个空集合你只能用 `set()` 而不能用 `{}` ，因为后者是创建一个空字典，这种数据结构我们会在下一节进行讨论。

以下是一些简单的示例

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # fast membership testing
True
>>> 'crabgrass' in basket
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                              # letters in a or b or both
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # letters in both a and b
{'a', 'c'}
>>> a ^ b                              # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

类似于 列表推导式，集合也支持推导式形式

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

5.5. 字典

另一个非常有用的 Python 内置数据类型是 *字典* (参见 映射类型 --- dict)。字典在其他语言里可能会被叫做 *联合内存* 或 *联合数组*。与以连续整数为索引的序列不同，字典是以 *关键字* 为索引的，关键字可以是任意不可变类型，通常是字符串或数字。如果一个元组只包含字符串、数字或元组，那么这个元组也可以用作关键字。但如果

理解字典的最好方式，就是将它看做是一个 *键: 值* 对的集合，键必须是唯一的（在一个字典中）。一对花括号可以创建一个空字典：`{}` 。另一种初始化字典的方式是在一对花括号里放置一些以逗号分隔的键值对，而这也是字典输出的方式。

字典主要的操作是使用关键字存储和解析值。也可以用 `del` 来删除一个键值对。如果你使用了一个已经存在的关键字来存储值，那么之前与这个关键字关联的值就会被遗忘。用一个不存在的键来取值则会报错。

对一个字典执行 `list(d)` 将返回包含该字典中所有键的列表，按插入次序排列 (如需其他排序，则要使用 `sorted(d)`)。要检查字典中是否存在一个特定键，可使用 `in` 关键字。

以下是使用字典的一些简单示例

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

`dict()` 构造函数可以直接从键值对序列里创建字典。

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

此外，字典推导式可以从任意的键值表达式中创建字典

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

当关键字是简单字符串时，有时直接通过关键字参数来指定键值对更方便

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

5.6. 循环的技巧

当在字典中循环时，用 `items()` 方法可将关键字和对应的值同时取出

```
...     print(k, v)
...
gallahad the pure
robin the brave
```

当在序列中循环时，用 enumerate() 函数可以将索引位置和其对应的值同时取出

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe
```

当同时在两个或更多序列中循环时，可以用 zip() 函数将其内元素一一匹配。

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}?  It is {1}.'.format(q, a))
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

如果要逆向循环一个序列，可以先正向定位序列，然后调用 reversed() 函数

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
3
1
```

如果要按某个指定顺序循环一个序列，可以用 sorted() 函数，它可以在不改动原序列的基础上返回一个新的排好序的序列

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print(f)
...
apple
banana
orange
pear
```

有时可能会想在循环时修改列表内容，一般来说改为创建一个新列表是比较简单且安全的

```
>>> import math
```

```
>>> for value in raw_data:
...     if not math.isnan(value):
...         filtered_data.append(value)
...
>>> filtered_data
[56.2, 51.7, 55.3, 52.5, 47.8]
```

5.7. 深入条件控制

while 和 if 条件句中可以使用任意操作，而不仅仅是比较操作。

比较操作符 in 和 not in 校验一个值是否在（或不在）一个序列里。操作符 is 和 is not 比较两个对象是不是同一个对象，这只对像列表这样的可变对象比较重要。所有的比较操作符都有相同的优先级，且这个优先级比数值运算符低。

比较操作可以传递。例如 a < b == c 会校验是否 a 小于 b 并且 b 等于 c。

比较操作可以通过布尔运算符 and 和 or 来组合，并且比较操作（或其他任何布尔运算）的结果都可以用 not 来取反。这些操作符的优先级低于比较操作符；在它们之中，not 优先级最高， or 优先级最低，因此 A and not B or C 等价于 (A and (not B)) or C。和之前一样，你也可以在这种式子里使用圆括号。

布尔运算符 and 和 or 也被称为 *短路* 运算符：它们的参数从左至右解析，一旦可以确定结果解析就会停止。例如，如果 A 和 C 为真而 B 为假，那么 A and B and C 不会解析 C。当用作普通值而非布尔值时，短路操作符的返回值通常是最后一个变量。

也可以把比较操作或者逻辑表达式的结果赋值给一个变量，例如

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

请注意 Python 与 C 不同，在表达式内部赋值必须显式地使用 海象运算符 := 来完成。 这避免了 C 程序中常见的一种问题：想要在表达式中写 == 时却写成了 =。

5.8. 比较序列和其他类型

序列对象通常可以与相同序列类型的其他对象比较。 这种比较使用 *字典式* 顺序：首先比较开头的两个对应元素，如果两者不相等则比较结果就由此确定；如果两者相等则比较之后的两个元素，以此类推，直到有一个序列被耗尽。 如果要比较的两个元素本身又是相同类型的序列，则会递归地执行字典式顺序比较。 如果两个序列中所有的对应元素都相等，则两个序列也将被视为相等。 如果一个序列是另一个的初始子序列，则较短的序列就被视为较小（较少）。 对于字符串来说，字典式顺序是使用 Unicode 码位序号对单个字符排序。 下面是一些相同类型序列之间比较的例子:

```
(1, 2, 3)              < (1, 2, 4)
[1, 2, 3]              < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
(1, 2, 3)             == (1.0, 2.0, 3.0)
```

注意对不同类型对象来说，只要待比较对象提供了合适的比较方法，就可以使用 < 和 > 来比较。例如，混合数值类型是通过他们的数值进行比较的，所以 0 等于 0.0，等等。否则，解释器将抛出一个 TypeError 异常，而不是随便给出一个结果。

备注

[1] 别的语言可能会返回一个可变对象，他们允许方法连续执行，例如 d->insert("a")->remove("b")->sort();。