5. 数据结构

本章深入讲解之前学过的一些内容，同时，还增加了新的知识点。

5.1. 列表详解

列表数据类型支持很多方法，列表对象的所有方法所示如下：

`list.append(x)`
在列表末尾添加一项。 类似于 `a[len(a):] = [x]`。

`list.extend(iterable)`
通过添加来自 iterable 的所有项来扩展列表。 类似于 `a[len(a):] = iterable`。

`list.insert(i, x)`
在指定位置插入元素。第一个参数是插入元素的索引，因此，`a.insert(0, x)` 在列表开头插入元素， `a.insert(len(a), x)` 等同于 `a.append(x)` 。

`list.remove(x)`
从列表中删除第一个值为 *x* 的元素。未找到指定元素时，触发 `ValueError` 异常。

`list.pop([i])`
移除列表中给定位置上的条目，并返回该条目。 如果未指定索引号，则 `a.pop()` 将移除并返回列表中的最后一个条目。 如果列表为空或索引号在列表索引范围之外则会引发 `IndexError`。

`list.clear()`
移除列表中的所有项。 类似于 `del a[:]`。

`list.index(x[, start[, end]])`
返回列表中 *x* 首次出现位置的从零开始的索引。 如无此条目则会引发 `ValueError`。

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)`
返回列表中元素 *x* 出现的次数。

`list.sort(*, key=None, reverse=False)`
就地排序列表中的元素（要了解自定义排序参数，详见 `sorted()`）。

`list.reverse()`
翻转列表中的元素。

`list.copy()`

返回列表的浅拷贝。 类似于 `a[:]`。

多数列表方法示例：

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.count('apple')
2
>>> fruits.count('tangerine')
0
>>> fruits.index('banana')
3
>>> fruits.index('banana', 4)  # 从 4 号位开始查找下一个 banana
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 `insert`, `remove` 或 `sort` 等仅修改列表的方法都不会打印返回值 -- 它们返回默认值 `None`。 [1] 这是适用于 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都可以排序或比较。 举例来说，`[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。 此外，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. 用列表实现堆栈

列表方法使得将列表作为栈来使用非常容易，最后添加的元素会最先被取出（“后时先出”）。 要将一个条目添加到栈顶，可使用 `append()`。 要从栈顶取出一个条目，则使用 `pop()` 而不必显式指定索引。 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. 用列表实现队列

列表也可以用作队列，最先加入的元素，最先取出（“先进先出”）；然而，列表作为队列的效率很低。因为，在列表末尾添加和删除元素非常快，但在列表开头插入或移除元素却很慢（因为所有其他元素都必须移动一位）。

实现队列最好用 `collections.deque`，可以快速从两端添加或删除元素。例如：

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry 到了
>>> queue.append("Graham")          # Graham 到了
>>> queue.popleft()                 # 第一个到的现在走了
'Eric'
>>> queue.popleft()                 # 第二个到的现在走了
'John'
>>> queue                           # 按到达顺序排列的剩余队列
deque(['Michael', 'Terry', 'Graham'])
```

5.1.3. 列表推导式

列表推导式创建列表的方式更简洁。常见的用法为，对序列或可迭代对象中的每个元素应用某种操作，用生成的结果创建新的列表；或用满足特定条件的元素创建子序列。

例如，创建平方值的列表：

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

注意，这段代码创建（或覆盖）变量 `x`，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)))
```

或等价于：

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

上面这种写法更简洁、易读。

列表推导式的方括号内包含以下内容：一个表达式，后面为一个 `for` 子句，然后，是零个或多个 `for` 或 `if` 子句。结果是由表达式依据 `for` 和 `if` 子句求值计算而得出一个新列表。 举例来说，以下列表推导式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)]
```

等价于：

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

注意，上面两段代码中，`for` 和 `if` 的顺序相同。

表达式是元组（例如上例的 `(x, y)`）时，必须加上括号：

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # 新建一个将值翻倍的列表
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # 过滤列表以排除负数
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # 对所有元素应用一个函数
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # 在每个元素上调用一个方法
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # 创建一个包含 (数字, 平方) 2 元组的列表
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # 元组必须加圆括号，否则会引发错误
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1
    [x, x**2 for x in range(6)]
     ^^^^^^^
SyntaxError: did you forget parentheses around the comprehension target?
>>> # 使用两个 'for' 来展平嵌套的列表
>>> 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，甚至可以是另一个列表推导式。

下面这个 3x4 矩阵，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],
... ]
```

下面的列表推导式可以转置行列：

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

如我们在之前小节中看到的，内部的列表推导式是在它之后的 for 的上下文中被求值的，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):
...     # 以下 3 行实现了嵌套的列表组
...     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，最好用内置函数替代复杂的流程语句。此时，zip() 函数更好用：

```
>>> 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 解包实参列表。

5.2. del 语句

可以按索引而不是按值从一个列表移除条目：即使用 del 语句。 这不同于返回一个值的 pop() 方法。 del 语句还可被用来从列表移除切片或清空整个列表（之前我们通过将一个空列表赋值给切片实现此功能）。 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` 也可以用来删除整个变量：

```
>>> del a
```

此后，再引用 `a` 就会报错（直到为它赋与另一个值）。后文会介绍 `del` 的其他用法。

5.3. 元组和序列

列表和字符串有很多共性，例如，索引和切片操作。这两种数据类型是 *序列*（参见 序列类型 --- list, tuple, range）。随着 Python 语言的发展，其他的序列类型也被加入其中。本节介绍另一种标准序列类型：*元组*。

元组由多个用逗号隔开的值组成，例如：

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # 元组可以嵌套：
>>> u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # 元组是不可变对象：
>>> t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # 但它们可以包含可变对象：
>>> v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

输出时，元组都要由圆括号标注，这样才能正确地解释嵌套元组。输入时，圆括号可有可无，不过经常是必须的（如果元组是更大的表达式的一部分）。不允许为元组中的单个元素赋值，当然，可以创建含列表等可变对象的元组。

虽然，元组与列表很像，但使用场景不同，用途也不同。元组是 immutable （不可变的），一般可包含异质元素序列，通过解包（见本节下文）或索引访问（如果是 `namedtuples`，可以属性访问）。列表是 mutable （可变的），列表元素一般为同质类型，可迭代访问。

构造 0 个或 1 个元素的元组比较特殊：为了适应这种情况，对句法有一些额外的改变。用一对空圆括号就可以创建空元组；只有一个元素的元组可以通过在这个元素后添加逗号来构建（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',    # <-- 注意末尾的逗号
>>> len(empty)
```

```
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

语句 `t = 12345, 54321, 'hello!'` 是 *元组打包* 的例子：值 `12345`, `54321` 和 `'hello!'` 一起被打包进元组。逆操作也可以：

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

称之为 *序列解包* 也是妥妥的，适用于右侧的任何序列。序列解包时，左侧变量与右侧序列元素的数量应相等。注意，多重赋值其实只是元组打包和序列解包的组合。

5.4. 集合

Python 还支持 *集合* 这种数据类型。集合是由不重复元素组成的无序容器。基本用法包括成员检测、消除重复元素。集合对象支持合集、交集、差集、对称差分等数学运算。

创建集合用花括号或 `set()` 函数。注意，创建空集合只能用 `set()`，不能用 `{}`，`{}` 创建的是空字典，下一小节介绍数据结构：字典。

以下是一些简单的示例

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # 显示重复项已被移除
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # 快速成员检测
True
>>> 'crabgrass' in basket
False

>>> # 演示针对两个单词中独有的字母进行集合运算
>>>
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # a 中独有的字母
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # 存在于 a 中但不存在于 b 中的字母
{'r', 'd', 'b'}
>>> a | b                              # 存在于 a 或 b 中或两者中皆有的字母
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # 同时存在于 a 和 b 中的字母
{'a', 'c'}
>>> a ^ b                              # 存在于 a 或 b 中但非两者中皆有的字母
{'r', 'd', 'b', 'm', 'z', 'l'}
```

与 列表推导式 类似，集合也支持推导式：

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

5.5. 字典

另一个常用的 Python 内置数据类型是 *字典* (参见 映射类型 --- dict)。 字典在其他编程语言中可能称为“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 *键* 来索引的，键可以是任何不可变类型；字符串和数字总是可以作为键。 元组在其仅包含字符串、数字或元组时也可以作为键；如果一个元组直接或间接地包含了任何可变对象，则不可以用作键。 你不能使用列表作为键，因为列表可使用索引赋值、切片赋值或 `append()` 和 `extend()` 等方法进行原地修改。

可以把字典理解为 *键值对* 的集合，但字典的键必须是唯一的。花括号 `{}` 用于创建空字典。另一种初始化字典的方式是，在花括号里输入逗号分隔的键值对，这也是字典的输出方式。

字典的主要操作是通过键来存储值并根据给定的键来提取值。 通过 `del` 也可以删除键值对。 如果你使用已存在的键进行存储，则与该键相关联的旧值将丢失。

通过下标操作 (`d[key]`) 提取不存在的键的值会引发 `KeyError`。 要避免在试图访问可能不存在的键时遇到这种错误，可改用 `get()` 方法，它会在字典不存在某个键时返回 `None` (或指定的默认值)。

对字典执行 `list(d)` 操作，返回该字典中所有键的列表，按插入次序排列 (如需排序，请使用 `sorted(d)`) 。检查字典里是否存在某个键，使用关键字 `in`。

以下是一些字典的简单示例：

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> tel['irv']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'irv'
>>> print(tel.get('irv'))
None
>>> 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()` 构造函数可以直接用键值对序列创建字典：

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

字典推导式可以用任意键值表达式创建字典：

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

关键字是比较简单的字符串时，直接用关键字参数指定键值对更便捷：

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

5.6. 循环的技巧

当对字典执行循环时，可以使用 `items()` 方法同时提取键及其对应的值。

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
```

在序列中循环时，用 `enumerate()` 函数可以同时取出位置索引和对应的值：

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

同时循环两个或多个序列时，用 `zip()` 函数可以将其内的元素一一匹配：

```
>>> 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.
```

为了逆向对序列进行循环，可以求出欲循环的正向序列，然后调用 `reversed()` 函数：

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

按指定顺序循环序列，可以用 `sorted()` 函数，在不改动原序列的基础上，返回一个重新的序列：

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

使用 set() 去除序列中的重复元素。使用 sorted() 加 set() 则按排序后的顺序，循环遍历序列中的唯一元素：

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

一般来说，在循环中修改列表的内容时，创建新列表比较简单，且安全：

```
>>> import math
>>> raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
>>> filtered_data = []
>>> 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` 条件句不只可以进行比较，还可以使用任意运算符。

比较运算符 `in` 和 `not in` 用于执行确定一个值是否存在（或不存在）于某个容器中的成员检测。运算符 `is` 和 `is not` 用于比较两个对象是否是同一个对象。 所有比较运算符的优先级都一样，且低于任何数值运算符。

比较操作支持链式操作。例如，`a < b == c` 校验 `a` 是否小于 `b`，且 `b` 是否等于 `c`。

比较操作可以用布尔运算符 `and` 和 `or` 组合，并且，比较操作（或其他布尔运算）的结果都可以用 `not` 取反。这些操作符的优先级低于比较操作符；`not` 的优先级最高， `or` 的优先级最低，因此，`A and not B or C` 等价于 `(A and (not B)) or C`。与其他运算符操作一样，此处也可以用圆括号表示想要的组合。

布尔运算符 `and` 和 `or` 是所谓的 *短路* 运算符：其参数从左至右求值，一旦可以确定结果，求值就会停止。例如，如果 `A` 和 `C` 为真，`B` 为假，那么 `A and B and C` 不会对 `C` 求值。用作普通值而不是

布尔值时，短路运算符的返回值通常是最后一个求了值的参数。

还可以把比较运算或其它布尔表达式的结果赋值给变量，例如：

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

注意，Python 与 C 不同，在表达式内部赋值必须显式使用 海象运算符 `:=`。 这避免了 C 程序中常见的问题：要在表达式中写 `==` 时，却写成了 `=`。

5.8. 序列和其他类型的比较

序列对象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，比较前两个对应元素，如果不相等，则可确定比较结果；如果相等，则比较之后的两个元素，以此类推，直到其中一个序列结束。如果要比较的两个元素本身是相同类型的序列，则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，则两个序列相等。如果一个序列是另一个的初始子序列，则较短的序列可被视为较小（较少）的序列。 对于字符串来说，字典式顺序使用 Unicode 码位序号排序单个字符。下面列出了一些比较相同类型序列的例子：

```
(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)
(1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

注意，当比较不同类型的对象时，只要待比较的对象提供了合适的比较方法，就可以使用 `<` 和 `>` 进行比较。例如，混合的数字类型通过数字值进行比较，所以，0 等于 0.0，等等。如果没有提供合适的比较方法，解释器不会随便给出一个比较结果，而是引发 `TypeError` 异常。

备注

[1] 别的语言可能会将可变对象返回，允许方法连续执行，例如 `d->insert("a")->remove("b")->sort();`。